Large-Scale Empirical Study of Important Features Indicative of Discovered Vulnerabilities to Assess Application Security

Mengyuan Zhang, Xavier de Carné de Carnavalet, Lingyu Wang, *Member, IEEE*, Ahmed Ragab

***Abstract*—Existing research on vulnerability discovery models shows that the existence of vulnerabilities inside an application may be linked to certain features, e.g., size or complexity, of that application. However, the applicability of such features to demonstrate the relative security between two applications is not well studied, which may depend on multiple factors in a complex way. In this paper, we perform the first large-scale empirical study of the correlation between various features of applications and the abundance of vulnerabilities. Unlike existing work, which typically focuses on one particular application resulting in limited successes, we focus on the more realistic issue of assessing the relative security level among different applications. To the best of our knowledge, this is the most comprehensive study of 780 real world applications involving 6,498 vulnerabilities. We apply seven feature selection methods to nine feature subsets selected among 34 collected features, which are then fed into six types of machine learning models, producing 523 estimations. The predictive power of important features is evaluated using four different performance measures. Our study reflects that the complexity of applications is not the only factor in vulnerability discovery, and that human related factors contribute to explaining the number of discovered vulnerabilities in an application.**

***Index Terms*—Software Vulnerability Analysis, Vulnerability Discovery Model, Software Security, Machine Learning**

I. Introduction

EXISTING research on vulnerability discovery models (VDMs) shows that vulnerabilities in an application may be linked to certain features, e.g., size or complexity, of that application (a more detailed review of related work will be given in Section VI). Those findings lead to an interesting question, i.e., *can we predict the existence of vulnerabilities in an application based on its features?* However, as shown in most existing works on VDMs, including those that focus on predicting vulnerable components inside one application [1], [2], [3], [4], [5], [6] and those that aim to establish mathematical models based on the historical vulnerability data of one application [7], [8], the correlation between vulnerabilities and the features of an application is usually not straightforward or reliable enough for such a prediction. For instance, the number of vulnerabilities in an application may not only be proportional to its size or complexity (as our study will show, no single feature is likely to have such prediction power). In other words, the existence of vulnerabilities in one particular application may depend on multiple factors in a complex way, which is still largely unknown.

In this paper, unlike existing works that typically focus on function/file/component level and construct models from one or a few applications, we focus on the more realistic issue of predicting the relative likelihood of vulnerabilities among a large number of applications. To this end, we perform, to the best of our knowledge, the first large-scale empirical study using machine learning techniques on the correlation between the abundance of vulnerabilities in an application and a rich collection of features. More specifically, our main contributions are as follows. To the best of our knowledge, this is the most comprehensive study to date involving 780 real-world software applications and 6,498 vulnerabilities. We apply seven feature selection methods on a rich collection of nine subsets of features that are then fed into six learning models. The predictive power has been evaluated using four different performance metrics including a correlation coefficient.

A. Overview

An overview of our study is given below, and is illustrated in Fig. 1.

- First, we obtain a large-scale dataset of open-source applications from GitHub, each of which involves at least one vulnerability listed in NVD [9]. In Section II, we address various challenges in the data collection, e.g., automatically identifying GitHub repositories, obtaining the accurate number of vulnerabilities and automating the labeling of repositories and mapping to products.
- Second, 34 features in total are extracted from the software applications. They fall into five different categories: popularity metrics, developer metrics, software property metrics, software metrics, and security metrics. To the best of our knowledge, this is the first effort to combine diverse features at the software level. Besides the commonly used features on code complexity and developer activity [10], we introduce popularity metrics to reflect the potential interest from attackers, software property metrics to indicate various intrinsic properties of the software, and security metrics to illustrate the potential attack surface of the applications. This is detailed in Section II-E.

M. Zhang, X. de Carné de Carnavalet, and L. Wang are with the Concordia Institute for Information Systems Engineering (CIISE), Concordia University, H3G 1M8, Montreal, QC, Canada. E-mail: {mengy_zh,x_decarn,wang}@ciise.concordia.ca.

A. Ragab with the Mathematics and Industrial Engineering Department, École Polytechnique de Montréal, C.P. 6079, Succ. Centre-Ville, H3C 3A7, Montreal, QC, Canada, and the Department of Industrial Electronics and Control Engineering, Faculty of Electronic Engineering, Menoufia University, Menouf, 32952, Egypt. E-mail: ahmed.ragab@polymtl.ca.

The list of extracted features for the 780 GitHub repositories is available upon request to the first author.

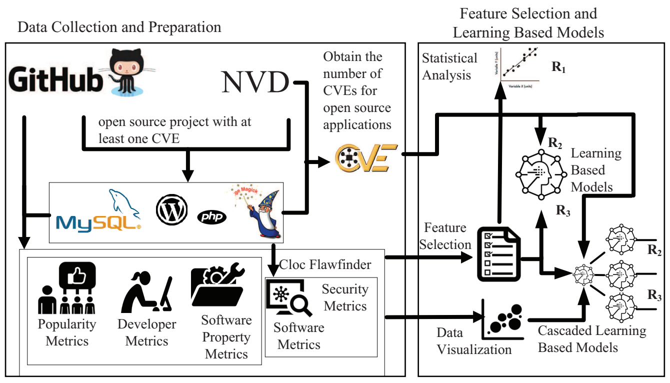

Data Collection and Preparation
Feature Selection and Learning Based Models
NVD
Obtain the number of CVEs for open source applications
open source project with at least one CVE
Statistical Analysis
R1
R2
Learning Based Models
R3
Feature Selection
Cloc Flawfinder
Security Metrics
Software Metrics
Popularity Metrics
Developer Metrics
Software Property Metrics
Data Visualization
Cascaded Learning Based Models

Fig. 1: An overview of the study

- Third, to reduce errors during the prediction process due to correlated or noisy features, we apply various feature selection and projection methods to extract nine subsets, including the original set of 34 features for reference and its projection using Principal Component Analysis (PCA) [11].
- Fourth, we study three research questions, as detailed in Section I-B, related to the predictive power based on the aforementioned features, using both statistical analysis and learning-based models. In our study, we apply two statistical analysis methods, namely, Pearson correlation and Kolmogorov-Smirnov test, to evaluate the discriminative power of features. Nine feature subsets (seven from feature selection results, one with the original set and one extracted from the PCA) are fed into six learning models to predict the number of vulnerabilities in the software. We provide detailed descriptions of the study and analysis of the prediction results in Section IV.
- Finally, we leverage t-SNE [12] to visualize our multi-dimensional dataset into two dimensions and observe separable clusters for applications in Section IV-C. We investigate the criteria behind this segregation and find that the size of the projects plays an important role in grouping applications. One of such groups is populated only by very low numbers of CVEs.

B. Research Questions and Our Main Findings

A common belief is that the complexity of an application is the main cause of vulnerabilities [3]. However, the reality is more complex since many other factors may come into play. For example, in our GitHub dataset, the project *opencms-core*,[1] 780,638 lines of code, which has a lower complexity than the project *SiberianCMS*,[2] 1,863,840 lines of code, actually turns out to have a higher number of vulnerabilities. Upon closer examination, we find that the project *opencms-core* was first released 2,331 days ago and project *SiberianCMS* was only released 601 days ago. The latter project contains a lower number of vulnerabilities, likely due to having a smaller time window for attackers to exploit. In this case, the complexity is clearly not the only determining feature. Similarly, we find through other examples that any type of features, e.g., the age (the days counted from the release time until now) or the popularity (the number of stars, watches, and forks), alone is not likely to yield reliable prediction power. Therefore, we set up three research questions and summarize our main findings as follows. We use #CVEs to represent *the number of CVEs* in the following sections.

- **R1:** *Is there just one feature that is significantly correlated with the number of vulnerabilities in different applications?*
  Our Finding: In the literature, a correlation coefficient of less than 0.3 corresponds to a weak correlation, 0.3–0.5 to a medium correlation, and greater than 0.5 means a strong correlation [3]. If we follow the same interpretation, only two features fall into medium correlation; the number of commits from our developer metrics and the application's age. The rest of the features are only weakly correlated to #CVEs. In the later discriminative test, all the features are rejected in the K-S test with a small *p-value*.
  The conclusion we draw from this research question is that instead of complexity, human factors share higher correlation with #CVEs. Indeed, the number of commits (from our developer metrics) and the application's age (from software property metrics, although it also indicates the attack windows for an application) share medium positive correlation with #CVEs. However, even highly correlated features follow a different distribution than that of #CVEs.
- **R2:** *Is there a combination of features that is significantly correlated with the number of vulnerabilities in different applications?*
  Our Finding: In our experiments, the subset of features that are selected based on the embedded methods with the Decision Tree (*DT*) and the Boosted Tree (*BT*) algorithms have relatively good performance metrics and accuracy, as detailed in Section IV. The best correlation coefficient based on the *DT* feature set is calculated as 0.875, and the best one based on the *BT* feature set is calculated as 0.845. Both feature sets are considered to be strongly correlated with #CVEs.
- **R3:** *Can machine learning methods be applied to those features effectively to predict the number of vulnerabilities in different applications?*
  Our Finding: The *BT* regression model yields the best results with the *DT* feature subset, and the overall accuracy is around 77% when the tolerance range is [-5,5], as detailed in Section IV. This could provide a rough indicator about the relative abundance of vulnerabilities. In the cascaded model analysis, we discover that the size of a project could also indicate the general trend of #CVEs.

The rest of the paper is organized as follows. Section II describes the data collection and preparation from both GitHub and NVD. Section III applies the feature selection techniques on the dataset to generate the pre-selected feature sets as the input for learning-based prediction models. Section IV analyzes the software vulnerability prediction models. Section V discusses our research questions, provides practical use cases and lists a number of limitations. Section VI reviews related work, and finally Section VII concludes this paper.

[1]https://github.com/alkacon/opencms-core
[2]https://github.com/Xtraball/SiberianCMS

TABLE I: Number of CVEs per year in the NVD database between January 2008–October 2017)

Year	*Total*	2017	2016	2015	2014	2013	2012	2011	2010	2009	2008
# CVE	67,294	8,784	9,152	7,717	8,247	6,098	5,524	4,600	5,072	4,955	7,145

II. Data Collection and Feature Extraction

In this section, we describe the collection of a dataset of open source applications that are affected by at least one vulnerability listed in NVD [9], along with features extracted from both the source code and metadata provided by GitHub.

Overview. We consider the GitHub open-source platform, as it allows us to easily locate and retrieve the source code of possibly many applications. We investigate the last 10 years of CVE vulnerabilities (from Jan. 2008 to Oct. 2017, inclusive) to search for applications found on GitHub. In total, we consider 67,294 CVEs. A one-year distribution is given in Table I. We identify and overcome several challenges that make the attribution of GitHub repositories to CVEs difficult. Finally, we build a semi-automated attribution tool that lifts more than half of the manual verification load. After identifying these repositories, we download their source code and extract metadata provided by GitHub to extract meaningful features.

A. Identifying GitHub repositories in CVEs

We consider the NVD database rather than the original MITRE CVE database since the former includes important additional information, e.g., the list of affected applications codified using the Common Platform Enumeration (CPE) dictionary. Within a CVE entry, several URLs are provided as references and may relate to, e.g., official statements about the vulnerability, advisory bulletins, proof-of-concept (PoC) or working exploits, links to a bug tracking system, or links to a patched difference. Among these references, we are interested in GitHub URLs; however, not all such URLs point to the official application repository that is affected by the CVE. To identify the correct repository, we could leverage the *reference_type* label provided on each URL. However, we found that unreliable, i.e., an official application repository could be labeled as *VENDOR_ADVISORY*, *PATCH* or *UNKNOWN*, without clear rules.

At this stage, we follow a conservative approach in identifying the official repository. Derived from our observations, we consider as official the first GitHub reference that is labeled as *VENDOR_ADVISORY* or *PATCH*, or which URL points to a specific *commit*, *issues*, or *pull*. Furthermore, we check whether the repository still exists and if it is not a fork of another GitHub repository. Non-existent repositories may have been a short-lived content related to the CVE, or they may simply indicate that the application is no longer hosted on GitHub. We do not consider forks due to the challenges in identifying whether the given CVE relates only to the fork or also to the forked application. We also verify that the given URL is not a simple advisory or a PoC by searching for keywords in the repository's name and description.[3] In parallel, we keep track of all GitHub repositories listed as references to help resolve certain conflicts in the next stage.

Another challenge is the change of repository owners or application names, making the same repository not uniquely identifiable across CVEs. Fortunately, GitHub redirects requests for the old URL to the new one. Hence, for each repository, we update its URL to the latest one, thus removing such discrepancies.

We found 5,737 CVEs that had a reference to a GitHub repository (official or not), accounting for 1,175 unique repositories, of which 24 no longer exist. 151 are redirected to a different repository (either due to a change in ownership or renaming of the application name), and 64 are identified as a PoC or other non-official repository according to our keyword filter. We identified 890 unique repositories as official applications corresponding to at least one CVE.

B. Challenges in obtaining the accurate number of CVEs per repository

Simply counting the occurrences of an identified GitHub repository across CVEs can be an unreliable indicator of the true #CVEs affecting the repository. In some cases, a CVE does not include a reference to a GitHub repository either because of a simple omission (e.g., RubyGems in CVE-2015-3900) or because the application's project did not exist on GitHub prior to a certain date. Moreover, a GitHub reference may be missing in several CVEs that affect the same repository, e.g., *tomhughes/libdwarf* is found only once while as many as 37 CVEs may be attributable according to affected products listed in all CVEs. A naive solution to missing GitHub references is to map each repository to the affected application listed in the CVEs where they are found, and count the CVEs where either the URL of the repository or the affected application is listed. For example, *tomhughes/libdwarf* can be mapped to *cpe:/a:libdwarf_project:libdwarf* as indicated in CVE-2015-8750.

Unfortunately, several discrepancies prevent us from simply aggregating CVEs for a given affected product: (a) Despite a codified list of affected products, the same product may be referred in various ways, e.g., *bestpractical/rt* can be listed as *cpe:/a:bestpractical:rt* or *cpe:/a:bestpractical:request_tracker*. Multiple duplicate applications need to be assigned to the same repository. (b) Also, CVEs with the correct affected product listed may only refer a GitHub repository that is not the one affected by the vulnerability (e.g., CVE-2017-13670). Yet, in other CVEs, the product may be tied to the right repository (e.g., CVE-2017-9609). This issue could lead us to attribute CVEs or repositories to the wrong product. (c) Finally, a repository could be associated with different products. For example, a

[3]We match the name of the repository with the following regular expressions (given in Python syntax): `([-_]vuln?$|vulnerabilit(y|ies)|advisor(y|ies)|exploits?|[-_]PoC$)` or `^(vulns?|CVE(-?.*)?$|PoC(-?.*)?$)` (case-insensitive), or `CVE|PoC|-SQLi|-XSS` (case-sensitive); we also match the description with `[Ee]xploits?|CVE\b|PoCs?\b|POCs\b|SQLi\b|XSS\b|\b[Bb]ug?s\b`.

family of products is built onto the same base, e.g., CVE-2017-0247 affects ASP.NET Core but refers to several Microsoft products (including ASP.NET Core). Also, the core project may not always be listed, further confusing attribution, e.g., CVE-2017-0028 lists Microsoft Edge as a vulnerable product, while it impacts ChakraCore, a core part of Edge's Javascript engine; at the same time, CVE-2017-8658 properly refers to ChakraCore as the affected product. Sometimes, both are referred under the same CVE, e.g., CVE-2017-11792. Furthermore, only better-known products could be referred as affected when only a depending library is affected, e.g., CVE-2017-2428 lists various Apple products but the vulnerability is located in nghttp2. In this case, we should only consider CVEs for nghttp2 as it corresponds to the repository listed, which is unaffected by other CVEs related to Apple products.

C. Semi-automation

We build a semi-automated tool that helps to label repositories and map them to affected products. It is a heuristic-based scoring system that assists the human expert to perform labeling by suggesting the most probable corresponding products for a given repository, and the rules to automatically attribute products to repositories and count CVEs when ambiguities are easily resolved.

1) Scoring system: After scanning all CVEs, we obtain a list of affected products for each CVE and a graph of related affected products (i.e., those listed in other CVEs that share one common affected product). We sort this list based on the similarity between the GitHub repository's owner/name and the affected product's organization/name. While performing manual labeling on a chunk of the dataset, we defined the similarity as a score. We evaluate the similarity between a repository and all affected products found in CVEs listing the repository plus their related products. Note that the list of products may expand quickly when it contains a popular product that is often found in CVEs related to its smaller parts or depending library.

For a given product-repository pair, we first compare the product organization with the repository owner (case-insensitive). Starting with a zero score, we give 4 points if the edit distance (Levenshtein distance) between both represents less than 10% of the longest string. This comparison encompasses small variations and can give high confidence that the organization and owner are the same entity. Else, we give 3 points if the Longest Common Substring (LCS) is longer than 60% of the longest string. This case is necessary when either the organization or the owner has an extra part appended, e.g, *Matroska-Org* vs. *matroska*.

Finally, if the pair passes neither the edit distance nor the LCS comparison, we break both strings around dashes, underscores and whitespace, and compare each subpart. For each pair of matching subparts (considering the edit distance and LCS criteria described above), we increase the score by two points weighted by the biggest proportion, in terms of subparts, that a matching subpart represents among both strings. Considering *sitaram_chamarty* vs. *sitaramc*, the matching subparts are *sitaram* (one subpart out of two) and *sitaramc* (one subpart out of one, which is the biggest proportion), so we increase the score by $2 \times 1/1$. This step takes into account names that are related as they share a common part, but are further apart. We show in Fig. 2 how these two points could be misleading when the application names are abbreviated in certain cases only.

We reproduce the same schema on the product's name compared to the repository's name; however, we strip any dashes, underscores and digits when comparing the whole names, as those are mostly noisy characters. Also, we attribute 5 points instead of 4 if the names agree with a small edit distance, which gives more importance to matching product/repository names than a matching organization. Moreover, we give an extra point if the product and repository names are exactly the same. This helps to break ties when very similar names get high scores, e.g., *openssl* vs. *openssh*. Finally, if the score is non-zero, we check whether the organization and product names are the same (considering the same edit distance and LCS criteria), e.g., *phpbb:phpbb3*. We give 2 additional points in this case since such names tend to match official products, by contrast to forks that carry a different organization name. The final score is rounded to the nearest integer. The maximum score is 12.

Products that get a score higher or equal to 5 are considered *best* matches. This threshold takes into consideration products that either receive 5 points directly thanks to a fully matching product/repository name, or by a combination of various levels of matching product vs. repository and owner vs. organization names, and/or benefit from the 2-point bonus for similar product and repository names. In any case, if such a product exists, it is strongly possible that it is the right one. At the same time, the threshold is low enough to capture products with e.g., owner/organization names that match partially (2 points), and a loosely comparable product/repository (3 points). Setting a higher threshold may miss some loosely related names, while setting it lower would encompass more unrelated names and yield many false positives.

2) Heuristics: Some situations can be resolved automatically. For instance, if a repository is assigned only one product across all its CVEs and this product is a best match, then we map the repository with this product and combine the CVEs. Also, if considering the CVEs attached to all the affected products does not make a difference from simply counting the occurrences of the GitHub repository being referred as the official one, then we stop the product mapping and output #CVEs. We also ignore repositories that specify the keyword "mirror" in their description, as such repositories do not live on GitHub and therefore the popularity metrics we can extract may be unreliable.

We force manual inspection on any repository that has fewer than 5 stars, no fork, and consists of 15 files or fewer. Such metrics indicate an unpopular repository that might not be an official application repository. For other repositories, the scoring system helps us to decide quickly among the best matched products. Among the 890 repositories we considered, 468 were labeled automatically, 21 were discarded as mirrors, 50 were removed due to: being deprecated (e.g., *horde/horde*), vulnerability finder or exploit generator (e.g., *rapid7/metasploit-framework*), PoC/advisories not caught by the previous filter (e.g., *Ha0Team/crash-of-sqlite3*), non-software, e.g., PDF, papers, documentation, websites (e.g., *nonce-disrespect/nonce-disrespect*), manually labeled mirrors/read-only repositories (e.g., *LibreOffice/core*), or empty repositories. Then, 27 repositories were ignored due to the complexity of properly counting their number of CVE. This happened when repositories are part of bigger projects, or a project spans on multiple repositories and CVEs do not label the specific affected repository, as well as in the case of forked companies, e.g., ownCloud and nextCloud. Further work is required to take such cases into account. Finally, four repositories were discarded since we could not obtain all metrics. In total, we assigned a more accurate #CVEs and we further consider 788 repositories.

D. Example

Fig. 2 shows an instance of our tool on a given repository. The upper half shows the identified candidate products by increasing score ("s:"). In the third position, *bestpractical:request_tracker* is mentioned directly in 16 CVEs ("#") and was attributed a score of 4 (due to matching organization/owner names). The star (*) before the name indicates that this product was seen in at least one CVE along with the GitHub repository. Below, `-this- (1)` indicates that the repository was actually seen exactly once. In second position with one CVE and a score of 6, the product *bestpractical:rt-extension-mobileui* scored higher due to two additional points attributed for the shared *rt* part in the product/repository name. However, this product was never mentioned in any CVE together with the repository. Having more than 5 points, this product is pre-selected (">" before the product) by the tool. After careful review, we found that this product corresponds to a mobile version of the main application that is located in a separate repository. Hence, we discard this product. The suggestion of this related product by our tool could have been useful in another situation. Finally, the best score corresponds to the product *bestpartical:rt* whose name fully matches the repository's name and hence receives 10 points (i.e., 5+4+1). It is used in 36 CVEs.

The lower half shows information related to the GitHub repository and the number of CVEs where a URL from this repository was seen, i.e., only in three CVEs in this case. The tool pre-selected the position of the products with at least 5 points. The operator changed this choice to those in positions 1 and 3. The tool merged the CVEs that belong to both selected products (16 and 36 CVEs) plus those in which the URL was found (3 CVEs, of which one appears with the 3rd product and two with the 1st product), giving a total of 52 CVEs. In this case, there was no overlap between the CVEs linked to both products; however, our tool would properly account for it if any. This number could not be obtained by counting either only the number of CVEs in which the repository's URL is found, or by mapping only one product name to the repository.

E. Feature Extraction

We clone the 788 GitHub repositories that are shortlisted. We also proceed to retrieve certain metadata from GitHub

```
3:  *cpe:/a:bestpractical:request_tracker (#16,s:4)
-this- (1)
2: > cpe:/a:bestpractical:rt-extension-mobileui (#1,s:6)
-
1: >*cpe:/a:bestpractical:rt (#36,s:10)
-this- (2)

https://github.com/bestpractical/rt
CVEs: 3
Request Tracker, an enterprise-grade issue tracking system
[1,2]? 1,3
Merged CVEs: 52
```

Fig. 2: Output of our semi-automated tool asking the operator to choose which product corresponds to the GitHub repository *bestpractical/rt*

TABLE II: The overview of GitHub applications features

Metrics	Features	Variables
Popularity	Number of stars	*#stars*
	Number of watches	*#watches*
	Number of forks	*#forks*
Developer	Number of contributors	*#contributors*
	Number of commits	*#commits*
Software Property	Age	*age*
	Number of labels	*#labels*
	Language distribution	%+language, e.g., *%Java*
Software	Size	*size*
	Number of files	*#files*
	Number of program files	*#program-files*
	Number of blank lines	*#blank*
	Number of comment lines	*#comment*
	Number of code lines	*#code*
	Number of lines of C/C++	*c-sloc*
Security	Number of issues	*#issues*
	Number of functions	*#functions*
	Flawfinder risk levels	*hits*, *L1–L5*

either through the provided APIs or by parsing relevant elements from web pages of the repository. We detail below the categories of features that we extracted from GitHub and from the cloned source codes. Table II shows an overview of our extracted features. Note that eight repositories systematically crashed the tools we used to extract certain features. In the end we only collected features for 780 repositories.

1) Popularity Metrics: The popularity metrics translate the incentives attackers or defenders may have to find vulnerabilities in a given project; the higher the popularity, the more attention to the project. In this study, the popularity metrics are queried from GitHub's API, i.e., *fork*, *star*, and *watch*. *#forks* reveals the popularity of a repository among active developers. Developers/contributors could work on a forked project, e.g., making additions or fixing bugs. Later, they could send a *pull* request to the original owner to include their modifications into the original code base. *#stars* and *#watches* on the project show the attention of a repository among interested GitHub users. *Star*ring a project is similar to bookmarking as it allows users to quickly reach back to the repositories. *Watch*ing a project enables users to receive notifications about the project.

2) Developer Metrics: We collect *#commits* during a project's existence on GitHub, as well as *#contributors*. *#commits* represents the number of changes/patches that have been made, and could also represent the overall development speed or trend (i.e., bigger vs. smaller changes committed at a time) over the lifespan of the project. This number is not directly available through the GitHub API; rather, one should query the metadata of all commits and count them, generating several hundreds or thousands of queries. Instead, we query the GitHub repository web page and parse *#commits* shown.

#contributors may relate to the size of a project in terms of manpower. It could also highlight open-source projects that accept various inputs from other developers, which could translate into varying levels of scrutiny on each contribution. We obtain this number through the GitHub API.

3) Software Property Metrics: We consider the creation date of a repository, which we translate to an *age* in days relative to the day we collected other time-dependent features in November 2017. Arguably, the *age* of a project is expected to make a difference in #CVEs that have been discovered, as it reflects the exposure period of the application, i.e., the time anyone has to see the source code, study the application, and discover vulnerabilities.

We also consider the relative percentage of each programming language in a repository, calculated based on the summed file sizes corresponding to each language. This distribution is useful considering the diversity of applications within scope as it puts the number of lines of code into perspective. Indeed, ten lines of Python arguably carry a different amount of information than ten lines of C, due to Python being a higher-level language. Up to 162 languages are reported by the API in our study; however, we only consider the 12 most common ones encountered in the repositories: C, C++, C#, Go, HTML, Java, JavaScript, Perl, PHP, Python, Ruby, and Shell, to limit the dimension of the features.

Finally, we consider the topics assigned to the repository by the owner. Topics are labels that usually reflect the purpose, subject area, functionalities, community and language of the repository. Examples include “cloud”, “design”, “ecommerce”, “framework”, “packet-capture”, “exploitation”, “windows”, “forum”, “php.” Due to the wide variety of topics and the lack of a proper hierarchical structure to group related topics, we limit our use of topics in this study to their number. A repository that is labeled with several topics may serve multiple purposes and touch several areas, giving some indications about its source code. We refer to this feature as *#labels*.

4) Software Metrics: The software metrics we collected from the source code cover four levels of granularity: overall program *size*, *#files*, number of program files (*#program-files*), and the source lines of code (*#code*, an estimation for the *cyclomatic complexity* [13]).

To measure a project’s size, one option is to rely on the GitHub API, which provides a *size* parameter for each repository. However, the reported size reflects the server-side storage requirement for all revisions with certain storage optimization. Thus, we resort to cloning all projects locally and measure only the size of the HEAD tree, i.e., the view of the latest revision’s files tree. All projects occupy a total of 106GiB on disk, while the API reported only 73.5GiB. The total size representing all latest revisions is 31GiB. The size on disk seems to represent the one reported by the API plus the space taken by the current view of the tree.

#files is obtained by *git ls-files*, which lists files under the current repository exhaustively. We use cloc [14] to report *#code*, *#blank* and *#comment* in multiple programming languages. In total, the time effort spent on gathering those features from the 780 code repositories is 1.5 days. In addition, we also ran Flawfinder [15] to gather C/C++ specific information, in particular the number of lines of C/C++ code, referred as *c-sloc*.

5) Security Metrics: Security metrics that we choose in this study represent two security perspectives: the potential attack likelihood, and the existing attack likelihood. We use the number of flaws that could be identified by Flawfinder, a widely used source code auditing tool [6], [15], as the potential attack likelihood for an application. Flawfinder identifies potential flaws inside C/C++ source code and outputs the total number of flaws along with a breakdown according to five severity levels, e.g, *L1* corresponds to the number of flaws with a level 1 severity (the lowest).

To quantify the potential threats in an application, we leverage the attack surface [16], which is defined as the sum of entry/exit points (i.e., the functions directly/indirectly invoking I/O functions). However, to obtain the attack surface, we would need to construct call graphs for all applications, which is not necessary trivial to establish. Therefore, we consider the number of functions, which is the upper bound of the attack surface metric, as the approximation of the attack surface metric to indicate the potential attack likelihood. The summation of the functions was indexed by *C-tag*.[4]

The number of issues on a project reflects software bugs reported by users and also list tasks for project maintainers. Certain issues are related to security, e.g., issue #6599 in *openssl*, a bug related to accepting invalid certificate versions,[5] which could lead to security vulnerabilities. In this study, the number of issues on a project is considered as an indicator of the existing attack likelihood.

III. Feature Selection

In this section, we apply machine learning techniques on our feature set to remove noisy and correlated features to the target variable, #CVEs. All the experiments are built with MATLAB. We uniformize the terms used in the latter sections. We refer to the target variable in regression models as *response* and the features in regression models are referred as *predictors*. The data entries are referred to as *observations* in all the models. Table III summarizes the feature selection results.

A. Feature Selection

A commonly known effect in machine learning, *curse of dimensionality*, points out that an increasing feature space dimensionality weakens the reliability of trained analysis systems [17] by *overfitting* the data. An efficient solution is to apply *feature selection* to find feature subsets, with lower-dimensional space, which leads to more reliable learning results. Feature selection is also known to enhance the prediction performance, lower the computational costs, simplify the models and provides better understanding in the learning problems [18]. We use three types of feature selection methods: filter methods, wrapper methods, and embedded methods.

[4]http://bxr.su/FreeBSD/usr.bin/ctags/
[5]https://github.com/openssl/openssl/issues/6599

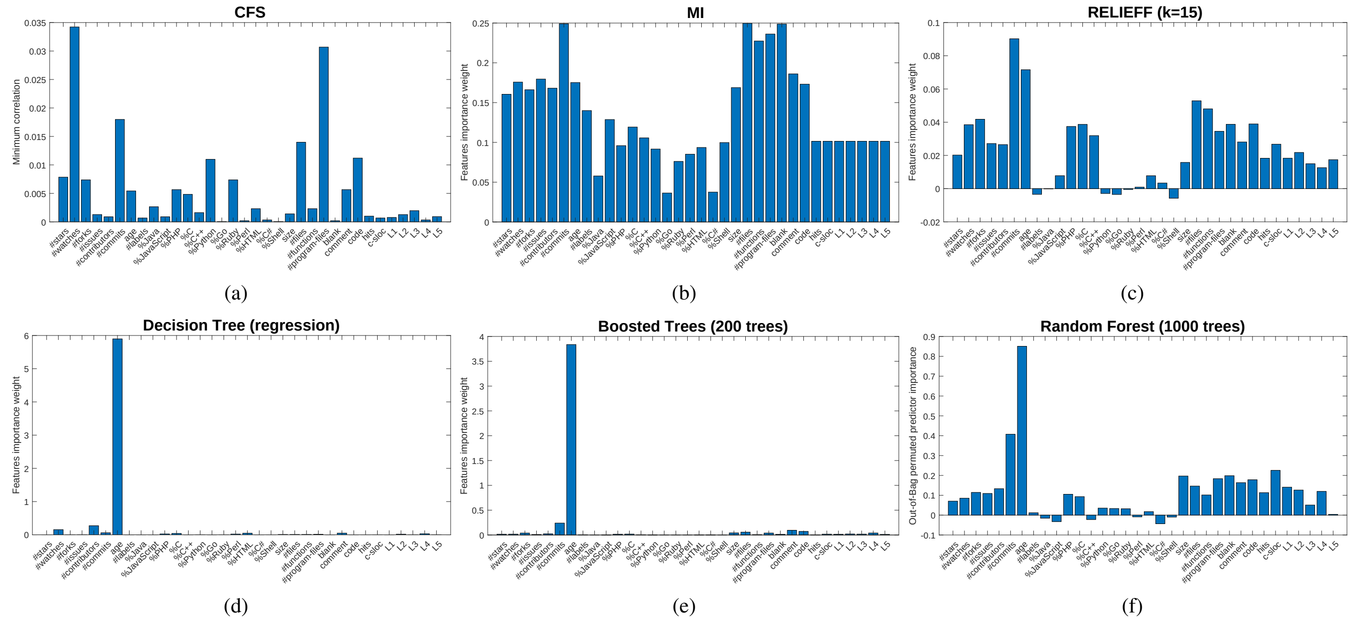


Fig. 3: Importance of our features according to six feature selection methods

TABLE III: Feature selection with different algorithms

			Methods						
			Filter			Wrap.	Embedded		
	Variables	Correlation	CFS	MI	ReliefF	SFS	DT	BT	RF
Popular.	#stars	0.0816		✓	✓				
	#watches	0.1420		✓	✓		✓		
	#forks	0.1425	✓	✓	✓	✓		✓	✓
Dev.	#contributors	0.1307	✓	✓	✓		✓	✓	✓
	#commits	0.4360		✓	✓		✓	✓	✓
Software Property	age	0.3363	✓	✓	✓		✓	✓	
	#labels	-0.0054							
	%Java	-0.0358				✓			
	%JavaScript	-0.0202							
	%PHP	0.0499			✓				✓
	%C	0.0553			✓			✓	
	%C++	0.0319			✓				
	%Python	-0.0479							✓
	%Go	-0.0204							
	%Ruby	-0.0421							
	%Perl	-0.0047				✓			
	%HTML	0.0072				✓			
	%C#	-0.0119							
	%Shell	-0.0279				✓			
Software	size	0.1266	✓	✓				✓	
	#files	0.2600		✓	✓			✓	✓
	#program-files	0.1620		✓	✓			✓	✓
	#blank	0.1956	✓	✓	✓				✓
	#comment	0.1694	✓	✓	✓			✓	✓
	#code	0.2036		✓	✓			✓	✓
	c-sloc	0.1966	✓		✓				✓
Security	#issues	0.0805		✓	✓				✓
	#functions	0.2658	✓	✓	✓				✓
	hits	0.1246	✓						✓
	L1	0.1209	✓						✓
	L2	0.1532	✓		✓			✓	✓
	L3	0.1001	✓						✓
	L4	0.0521						✓	✓
	L5	0.1005	✓			✓			

In total, we used seven feature selection methods to obtain good feature subset candidates. Additionally, we combine 34 dimensions by using PCA to obtain the eighth feature subset.

1) Filter Methods: The filter methods evaluate the score of each feature according to certain criteria. The experts then choose the subsets based on the scores. We consider the correlation-based feature selection (*CFS* [19]), and the mutual information (*MI* [20]) methods. However, filter methods only consider the relationship between the pairs of features; the relationships between multiple features are ignored.

We choose *CFS*, *MI* and *ReliefF* [21], as these algorithms are widely used in the literature.

a) Correlation based Feature Selection: CFS calculates the correlations between any pair of features, and uses the lowest correlation in one feature as the weight of this feature. The *CFS* method selects the features that highly correlate with the response and do not correlate with other features. Fig. 3a presents the feature selection scores from CFS. In this study, we first set 0.01 as the threshold for the *CFS* method, i.e., weights lower than 0.01 are filtered in this step. Then, only the features with high correlations with #CVEs (see the correlation column in Table IV) stay in the final feature set. In this sense, we obtain a feature set with high correlation to #CVEs and low correlation with any other features.

b) Mutual Information: The *MI* measures the mutual dependencies between the response and the predictors. As with the correlation, the *MI* algorithm only produces pairwise results. In contrast to the correlation, the *MI* algorithm captures the nonlinear dependency through the joint probabilities. In this study, we output the *MI* score as the importance weight for features. Fig. 3b shows the importance of the features as defined by their *MI* score. We set the threshold to select the feature subset to 0.15.

c) ReliefF: The *ReliefF* algorithm calculates the Euclidean distance from each predictor to the response. k is the number of closest predictors that are taken into consideration for the majority vote. We applied different values of k to generate the importance weights for the features, e.g., $k = 1$, 3, 5, 7, 15. Fig. 3c shows the feature selection results from $k = 15$, and in this study, we select the features from the $k = 15$, with a threshold of 0.02.

2) Wrapper Methods: The wrapper methods involve learning algorithms to evaluate the relevance of the feature sets. Ideally, wrapper methods test all the possible permutations for the feature subsets and output the ones with the best results in terms of accuracy. The corresponding computation time grows exponentially with the number of features. Heuristic algorithms, such as the sequential forward selection (*SFS*), are proposed to tackle this search problem. *SFS* starts from an empty set and adds features one by one to obtain the best feature set. We apply the *SFS* and the sequential forward floating selection (*SFFS*). We only report the result from *SFS* since both yield the same results.

3) Embedded Methods: The embedded methods build the learning algorithms inside the feature selection process, e.g., decision trees [22], random forests [23]. The selected feature set is generated automatically after the learning process.

We implemented three embedded methods: binary regression decision tree (*DT*), boosted regression trees using least squares boosting (*BT*), and Random Forest (*RF*). Fig. 3d demonstrates the importance weight for each feature from the *DT* method. We choose 0.05 as our threshold; another threshold value might be chosen base on expert knowledge.

The number of trees is the common parameter in both the *BT* and *RF* method; we choose 200, 500, and 1000 trees for both methods. During our experiment, changing the parameter's value only bring negligible differences in the feature importance weights. We only show the results from the *BT* method with 200 trees in Fig. 3e, and the *RF* method with 1000 trees in Fig. 3f. Based on expert knowledge, we choose threshold as 0.018 and 0.1, respectively.

IV. Analysis

In this section, we first apply two statistical methods to evaluate the correlations between the features and #CVEs in Section IV-A. Then, the prediction powers of learning-based models are analyzed in Section IV-B. Finally, we conduct cascaded model analysis to further study the relationships between features and #CVEs in Section IV-C.

A. Statistical Analysis

We apply two statistical methods to evaluate the correlations between our features and #CVEs. First, we normalize the feature sets with Equation 1 (*Min-Max standardization*) to transfer the values of features to a bounded range.

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{1}$$

where $x = (x_1, \ldots, x_n)$ are the original features in the dataset and z_i is the i^{th} normalized data. Equation 1 maps the values of all features into the same range $[0, 1]$.

The first statistical analysis is the Pearson coefficient, which illustrates the linear relationships between the response and predictors, e.g., in our case, the #CVEs and features. The correlation coefficient takes values between -1 to 1, which corresponds to a perfect direct decreasing (negative) or increasing (positive) linear relationship, respectively. The value 0 means that the two input variables are not correlated.

TABLE IV: Results of the statistical analysis for GitHub dataset based on Spearman's rank correlation coefficient and K-S test (significant with p-value ≤ 0.00029)

	Variable	Correlation	p-value	K-S test
Popularity Metrics	#stars	0.0816	9.116E-67	Reject
	#watches	0.1420	1.725E-71	Reject
	#forks	0.1425	5.385E-66	Reject
Developer Metrics	#contributors	0.1307	1.027E-68	Reject
	#commits	**0.4360**	1.676E-69	Reject
Software Property Metrics	age	**0.3363**	1.428E-302	Reject
	#labels	-0.0054	2.526E-40	Reject
	%Java	-0.0358	6.851E-72	Reject
	%JavaScript	-0.0202	1.004E-19	Reject
	%PHP	0.0499	8.033E-25	Reject
	%C	0.0553	6.826E-24	Reject
	%C++	0.0319	4.305E-37	Reject
	%Python	-0.0479	1.446E-31	Reject
	%Go	-0.0204	1.314E-96	Reject
	%Ruby	-0.0421	3.316E-62	Reject
	%Perl	-0.0047	1.043E-63	Reject
	%HTML	0.0072	3.822E-20	Reject
	%C#	-0.0119	1.314E-96	Reject
	%Shell	-0.0279	5.564E-23	Reject
Software Metrics	size	0.1266	1.676E-69	Reject
	#files	0.2600	1.676E-69	Reject
	#program-files	0.1620	1.676E-69	Reject
	#blank	0.1956	1.676E-69	Reject
	#comment	0.1694	4.155E-69	Reject
	#code	0.2036	1.676E-69	Reject
	c-sloc	0.1966	2.291E-32	Reject
Security Metrics	#issues	0.0805	3.316E-62	Reject
	#functions	**0.2658**	9.820E-27	Reject
	hits	0.1246	2.983E-38	Reject
	L1	0.1209	2.218E-37	Reject
	L2	0.1532	1.000E-39	Reject
	L3	0.1001	8.338E-37	Reject
	L4	0.0521	7.738E-51	Reject
	L5	0.1005	3.555E-51	Reject

The second method is the two-sample Kolmogorov-Smirnov test (K-S test),[6] which returns a decision and p-value, to demonstrate whether or not the response and the predictor are from the same continuous distribution. The Bonferroni correction method is used to deal with the multiple comparisons problem [24]. Therefore, in this paper, the significance level is corrected to 0.00029 = $\frac{0.01}{34}$ (corresponding to a non-corrected $p \leq 0.01$ for each test).

The top three correlated features to #CVEs are *#commits*, *age*, and *#functions* corresponding to the developer metric, software property metric and security metric. We discover that 32/34 features are weakly correlated to #CVEs. Only *#commits* and *age* fall in a medium correlation range (i.e., a correlation between 0.3 and 0.5). After carrying out the K-S test, all features are rejected due to the low p-value, which means that no feature follows same distribution as #CVEs.

In conclusion for **R1**, #commits is the feature that has the highest correlation to #CVEs; however, it is still under a medium correlation and not a strong one. Overall, software property metrics have the lowest correlation, e.g, most of the language distributions share weak negative correlation with #CVEs. Also, features in software metrics correlate with #CVEs around similar value, all of them show weak positive correlation. Security metrics and popularity metrics also show

[6]This test calculates the distance between the distribution of two samples; the null hypothesis is that the two samples are from the same distribution

weak correlations. However, all the features are under different distribution with #CVEs.

B. Single-model Learning-based Analysis

1) Experiment Models and Evaluation Methods: Unlike some existing works, which focus on vulnerability discovery at the file level within an application and with a binary output (i.e., vulnerable or not), this study leverages regression models with which responses are numeric values that correspond to #CVEs *per application*. We conducted this experiment using six regression models with various parameters on different sets of features, totaling 523 prediction results. We refer as *feature-selection-method* set the feature subset generated from the *feature-selection-method*, e.g., *DT* set.

Data Preparation. 240 out of 780 projects contain at least 50% of C/C++, and up to 302 contain any proportion of C/C++ in our dataset. As Flawfinder only supports C/C++ projects, we assign the value -1 to all Flawfinder-related features (expectedly positive integers) for projects with a zero amount of C/C++ according to the GitHub repository's metadata [25].

Performance metrics. The performance of a learning model is evaluated through root mean squared error (RMSE), a widely used measure of error that emphasizes large errors; mean absolute error (MAE), which measures the absolute difference between predicted values and responses; and mean absolute percentage error (MAPE) that gives a relative measure of discrepancies and also gives more emphasis on errors for small #CVEs. Usually, a lower rate of errors corresponds to better performance of a model for a given feature set. In addition, correlation is used to compare the trend between responses and predictors. In Table VI, the best feature sets are chosen based on the criteria given below.

Boosted Tree. We leveraged boosted regression trees using least squares boosting, *BT* and *BT-opt*. *BT-opt* is selected to calculate parameters with the inbuilt algorithms. Table V demonstrates the evaluation results of RMSE, MAE and MAPE. The prediction results from *BT-opt* with all features has the lowest MAPE; however, the RMSE and MAE are higher than *BT* with the *DT* set. The main reason for this observation is that evaluation methods are more sensitive to capturing errors in certain types of data. For example, MAPE is sensitive to the errors from small values inside one dataset while RMSE captures the existence of large errors resulting from the prediction.

Decision Tree for Regression. In the *DT* model, the *MI* set yields the best MAE; however, the *BT* set has the best RMSE and MAPE. Therefore, the prediction results from *DT* with *BT* set are presented in Table VI and Fig. 4b. The accuracy of the predicted results are presented in Fig. 5b.

Linear Regression. In the *LR* model, the *DT* set has the best RMSE. The *SFS* set has the best MAE and MAPE. The correlation is calculated as 0.451 and 0.434 from the predictions, respectively; therefore, we only demonstrate the best results from *DT* set in Table VI and show the predicted results and the accuracy in Fig. 4c and Fig. 5c.

Neural Networks. We apply two Neural Network (*NN*) models, function fitting NN (FFNN) and generalized regression NN (GRNN). To obtain relatively better results from both models, we choose 36 hidden layers (from 1 to 36) to get the best RMSE, MAE and MAPE for different feature sets. The *BT* set with 4 hidden layers obtains the best RMSE, and the *DT* set with 6 hidden layers obtains the best MAE and MAPE in the FFNN model, which is overall better than the GRNN model. Since the correlation is 0.533 for the *BT* set, which is higher than the correlation from the *DT* set, 0.406, we only show the result of the *BT* set in Table VI and show the predicted results and the accuracy in Fig. 4d and Fig. 5d.

Random Forest. We consider the *RF* model with various number of trees, 10, 100, 200, 500 and 1000. The best result is obtained from the *DT* set with 500 trees, and is presented in Table VI, Fig. 4e and Fig. 5e.

Support Vector Machine Regression. We leverage Support Vector Machine Regression (SVR) with three different solvers, i.e., ISDA, L1QP, and SMO solver, and with three kernels, i.e., Gaussian, Linear and Polynomial. The best overall predictive results are generated from the ISDA solver with the Gaussian kernel with the *DT* set; see Table VI. However, the *SVR* model also yields the lowest MAPE (66.61%) with the L1QP solver and polynomial kernel using the *CFS* set. We list the prediction results of the *CFS* set corresponding to the accuracy in Fig. 4f and Fig. 5f.

Cross-validation. Each of the training and prediction experiments has been conducted using ten-fold cross-validation [26]. The same randomized folds are used with the *linear*, *NN*, and *RF* models (i.e., we perform 10 training-prediction segments based on the randomly pre-separated folds, and average the results), while folds for the *BT*, *DT*, and *SVR* models are selected internally in MATLAB.

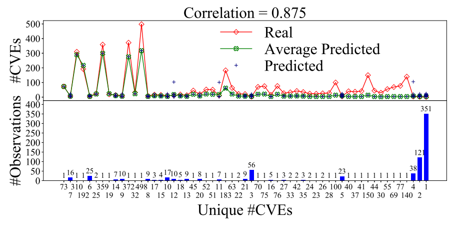
For the *NN* model, a validation set is also used to avoid overfitting [27]. We randomly split each training set (representing 90% of our data) into 9 parts and use one as the validation set, giving 80%, 10%, 10% for training, testing, and validation, respectively. These proportions are on par with MATLAB default's, i.e., 70%, 15%, 15% [27], but match more closely the proportions of a classic ten-fold cross-validation.

2) Experiment Results Interpretation: The *DT* set, which only contains four features from three categories, provides the best evaluation results for five models. It contains the most generic features, thus it discards all the application code specific features. This result indicates that the general comparison among applications could simply be generated from developers, popularity and the existing time of the applications. The *BT* set, which involves more software and security metrics, perform similarly to the *DT* set.

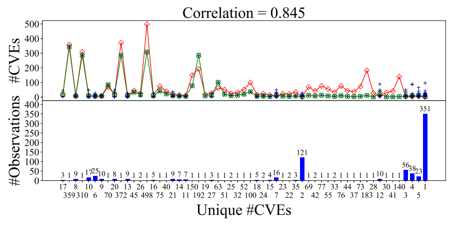
To better understand the prediction results, Fig. 4 summarizes the predicted value from the best models and the feature sets. Since multiple observations correspond to the same #CVEs in our dataset, we group the observations based on #CVEs. X-axes in the graphs represent bins of unique #CVEs. The red squares in the upper graph plot the real #CVEs for each unique #CVEs, which are the same as the labels on the x-axis. The green crosses are the average predicted #CVEs for each unique #CVEs and the blue pluses are the predicted #CVEs for each observation. The predicted results are ordered by the relative percentage error

TABLE V: Results of predictive power for *BT* and *BT-opt* regression models

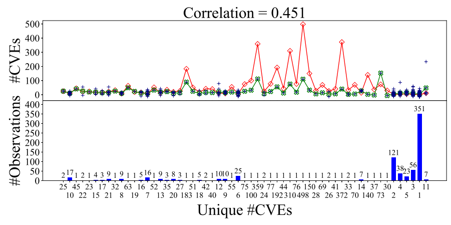
Model	**Performance Measures**	**All Features**	**Filter Method**			**Wrapper Method**	**Embedded Method**			**PCA**
			CFS	MI	RELIEF	SFS	**DT**	BT	RF	
BT	RMSE	16.57	17.23	16.57	16.57	31.72	**15.83**	16.57	31.1	25.77
	MAE	6.57	6.99	6.57	6.57	9.07	**6.37**	6.57	9.87	9.35
	MAPE (%)	206.91	218.71	206.91	206.91	263.69	**198.18**	206.91	291.24	291.51
BT-opt	RMSE	18.4	22.04	20.49	19.15	31.4	19.67	19.68	29.16	28.31
	MAE	6.92	7.76	7.61	6.78	9.19	6.96	6.86	8.47	9.81
	MAPE (%)	188.99	219.41	209.62	208.08	280.12	211.17	195.17	219.33	347.25



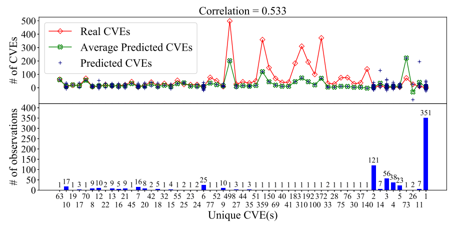
(a) *BT* with the *DT* set



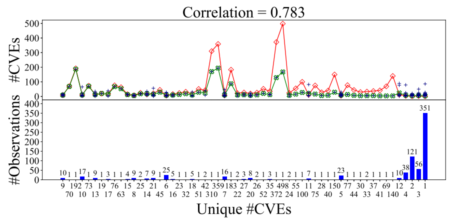
(b) *DT* with the *BT* set



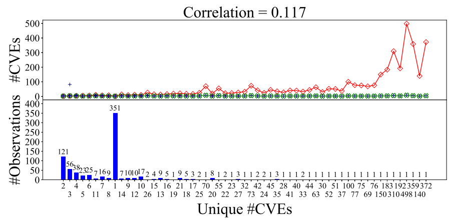
(c) *LR* with the *DT* set



(d) *NN* with the *BT* set



(e) *RF* with the *DT* set



(f) *SVR* with the *CFS* set

Fig. 4: Prediction results (x-axes in the sub-figures follow ascending order of the relative percentage error between average predicted #CVEs and real #CVEs)

TABLE VI: The best results of predictive power for 6 learning-based prediction models

	Models					
	BT	DT	LR	NN	RF-500	SVR-ISDA-Gaussian
	Best Feature Sets					
Performance Measures	DT	BT	DT	BT	DT	DT
RMSE	15.83	16.88	25.33	24.85	18.71	25.25
MAE	6.37	6.55	11.41	8.73	6.86	6.62
MAPE(%)	198.18	179.28	434.75	285.45	199.27	81.47
Correlation	0.875	0.845	0.451	0.533	0.783	0.117

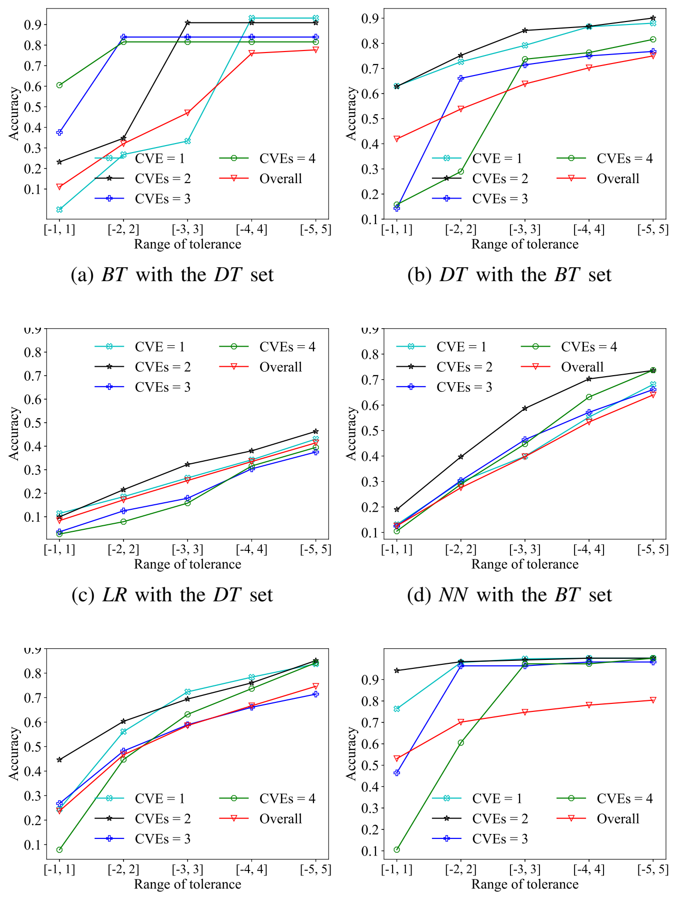
($\frac{|average\ predicted\ CVEs - real\ CVEs|}{real\ CVEs}$), for example in Fig. 4a the observation with CVEs equal to 73 associates with the smallest relative percentage error and the observations with #CVEs equal to 1 is the least relative accurate prediction in *BT* model with *DT* set. The correlation value between real #CVEs and the predicted #CVEs is 0.875 in this experiment. The higher correlation means that the trend of predicted #CVEs follows a similar trend with real #CVEs.

Results and Implications for Prediction Results from Regression Models: According to the predicted results from six different models, first we can observe that the predictions for each response vary significantly among models. For example, the best project (#CVEs = 73) in Fig. 4a is among the ones predicted with the largest errors in Fig. 4d. The majority of projects could be predicted to be closer to the original #CVEs within a certain tolerance range in different models, which is demonstrated in Fig. 5. In general, some projects with large #CVEs are well predicted by *BT* and *DT*, while the projects affected by small #CVEs are better predicted by *SVR* and the intermediate results are better when using *LR*, *NN* and *RF*.

Compared with the correlation results in Table IV, the correlation results between the predicted results and #CVEs from the learning-based models are improved significantly. This means that the *DT* and *BT* sets capture important factors in the software vulnerability discovery process.

Some projects have never been predicted close to their real #CVEs in the entire experiment. For instance, the project *the-tcpdump-group/tcpdump* is associated with 140 CVEs, which are never predicted accurately in our experiments. We observed that out of those 140 vulnerabilities, 132 of them were published during 2017 in two batches. In the batch from September 2017, 90 CVEs are disclosed by the same group of people,[7] suggesting that an automated process led to the numerous discoveries of similar code flaws. These rare but sudden mass disclosures suddenly and significantly increase the #CVEs for a project, which are not predicted by our model. As a result, there is a significant difference between the predicted #CVEs and the actual #CVEs. The best predicted value is 7 CVEs or this particular project from all the models. Without accounting for the mass disclosure in 2017, the predicted value matches closely the real remaining #CVEs

[7]https://usn.ubuntu.com/3415-2/



(a) *BT* with the *DT* set (b) *DT* with the *BT* set

(c) *LR* with the *DT* set (d) *NN* with the *BT* set

(e) *RF* with the *DT* set (f) *SVR* with the *CFS* set

Fig. 5: Accuracy for the predicted #CVEs with various ranges of tolerance

(i.e., 8). The project *ntp-project/ntp* is affected by 77 CVEs, and was also never predicted accurately. Similarly, we found that an unexpected large number of CVEs was released in 2017, i.e., 59/77. Our regression models could not capture the unexpected release of vulnerabilities due to human/automation factors, such as research projects or automated testing.

The relative percentage error is exaggerated among the small number of observations since a small difference already generates a large percentage error. For example, the relative error is 100% when the predicted number is 2 and the real number is 1. We study the relative range of tolerance to understand the predictive power in a small number of responses for different models. The x-axis of Fig. 5 shows the ranges of tolerance, e.g., [-1,1] means that we accept the prediction result with an error value of 1. The accuracy is defined as the percentage of accepted predictions. The lines on the figures show the change of accuracy with the increase of tolerance ranges.

Results and Implications for Accuracy: Fig. 5 demonstrates the predictive power for small #CVEs for each model based on the previously selected feature set. The red line in each graph shows the overall accuracy for the entire dataset. Since the most common value is #CVEs = 1, the overall accuracy is often close to the accuracy of this entry. Although the correlations are similar in *BT* and *DT* models, the predictions for small #CVEs are very different. The *DT* model, which has a lower MAPE than the *BT* model, illustrates the predictive power for small number of CVEs by having a relatively higher accuracy in #CVEs = 1 and 2. The performance of the *LR* and *NN* models is worse than the *BT* and *DT* models in this case. Fig. 5e shows the advantage of low MAE; the accuracy increases faster than other models. Fig. 5f shows the *SVR* model with SFS set, which has the lowest MAPE, with the best prediction results for #CVE = 1,2,3.

The *BT* and *DT* models provide a relatively better overall trend prediction, with more accurate predictions for larger #CVEs. The *LR* and *NN* models provide more accurate results for intermediate #CVEs. The *RF* model gives the best average increase in accuracy, thus could be used for the predictions with certain tolerance range. The *SVR* model could predict small values of the #CVEs in a more accurate way. The overall accuracy for the entire dataset reaches 77% with a [-5,5] tolerance range.

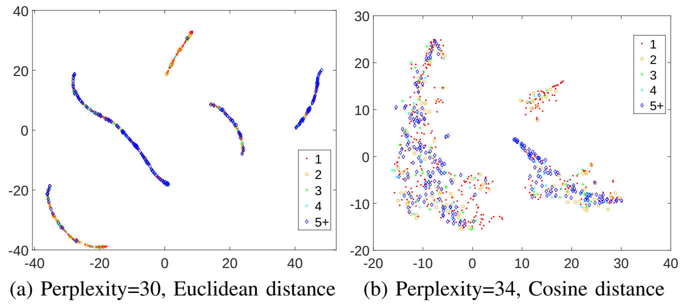
C. Cascaded Models

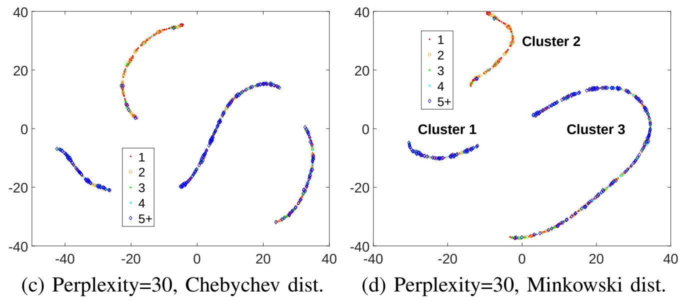
In this section, we visualize our all-feature dataset and identify clusters that correspond to specific #CVEs. Given that our single-model analysis yields models that perform better in certain range of #CVEs, we build a cascaded model by first distinguishing the clusters then applying the previous learning-based models.

The t-Distributed Stochastic Neighbor Embedding (t-SNE) [12] technique is used in this work to cluster our dataset and visualize it in two dimensions. t-SNE represents the similarities of high-dimensional datapoints as the conditional probabilities that are calculated by several algorithms. Compared to traditional dimensionality reduction techniques (e.g., PCA [11], which uses linear techniques), t-SNE performs better in keeping similar datapoints close together with nonlinear dimensionality reduction techniques, which cannot be achieved by linear mapping. Compared with other nonlinear dimensionality reduction techniques, such as Sammon mapping [28], Stochastic Neighbor Embedding (SNE) [29], t-SNE is capable of capturing both the local and the global structures of the datasets.

In MATLAB's t-SNE implementation [30], 11 distance algorithms can be chosen in the t-SNE function. The *Perplexity* parameter controls the effective number of local neighbors at each point. Fig. 6 is a visualization of the GitHub dataset using four distance algorithms. To visualize the data, we remove the response (#CVEs) from the dataset; only predictors are clustered by t-SNE. The 34 features are mapped into two dimensions. We then use #CVEs to color the data points. The blue dots represent applications affected by five or more #CVEs. The other colors correspond to the labeled #CVEs. In Fig. 6a, the datasets are visually well divided into five clusters; two of the clusters contain a majority of repositories with low #CVEs, and three of them contain a majority with higher #CVEs.

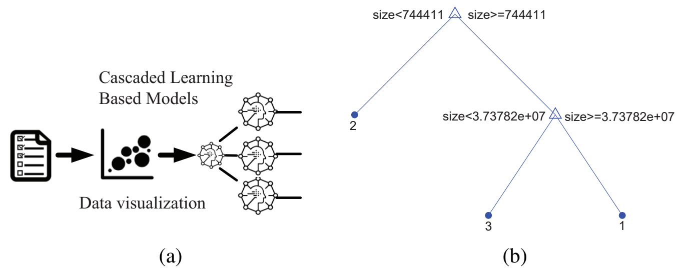
We applied all the distance algorithms with various perplexity values in order to obtain the best visualization results.


(a) Perplexity=30, Euclidean distance (b) Perplexity=34, Cosine distance



(c) Perplexity=30, Chebychev dist. (d) Perplexity=30, Minkowski dist.

Fig. 6: Visualization of our data using t-SNE projections based on all features, and selected perplexity and distance algorithms



(a) (b)

Fig. 7: Cascaded learning-based model (a), Decision tree to classify the clusters in Fig. 6d (b)

Fig. 6 shows the most visually separated visualization results from the GitHub dataset.

We design a new cascaded model (illustrated in Fig. 7a) to first separate the original dataset into three datasets according to Fig. 6d, namely cluster 1, 2 and 3. Cluster 1 is the lower left cluster in Fig. 6d; cluster 2 is the upper cluster; and cluster 3 is the lower right cluster. Then we apply machine learning classification techniques on the clusters to predict #CVEs for each project. In our dataset, after applying 10-fold cross validation, the decision tree classifier in Fig. 7b classify our dataset into three clusters with 100% accuracy. From the color of the clusters, we could notice that cluster 2 mainly contains small #CVEs (average #CVEs = 1.4); cluster 1 is mainly for higher #CVEs projects (average #CVEs = 24); and cluster 3 is mixed with both type of projects (average #CVEs = 7). This observation indicates that the size of a project corresponds to the #CVEs. Small sized projects always have low #CVEs; however, some large size projects only contain small #CVEs, which require other additional features to explain.

We applied all the models to the separated datasets, and compared the best evaluation methods and correlations. The best correlation between the predicted and real #CVEs in

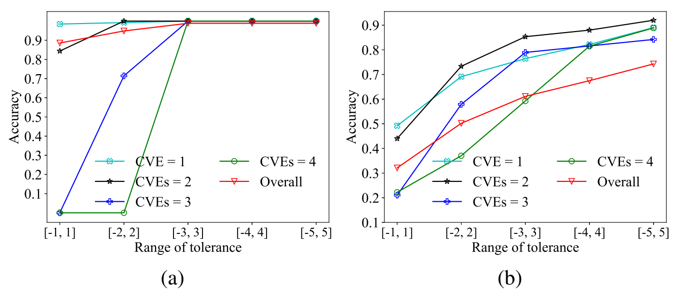


Fig. 8: Accuracy for cluster 2 and 3 from the cascaded model

cluster 1 is 0.9118, which indicates a better trend than the ones achieved in the single-model study. This observation shows that a large #CVEs dataset have the ability to compare the relative relationship between two large size applications. Cluster 2 has a lower MAE than in the single-model study (i.e., 0.38) using *SVR*, which demonstrates the ability of this model to accurately predict small #CVEs. This is consistent with our finding in the single-model study. The accuracy is shown in Fig. 8a, which has a 90% overall accuracy with a [-1,1] tolerance range. Cluster 3, which has mixed #CVEs, has shown a similar predictive power with the single-model study. We believe that more features or applying feature selection may increase predictive power for Cluster 3.

V. Discussion

In this section, we first address the three research questions, then we provide practical use cases for our proposed models. Finally, we list a number of limitations identified in this work.

A. Research questions

R1: After the statistical analysis of the dataset, we obtained the highest correlation value, 0.436, between *#commits* and #CVEs, which falls into a medium correlation range. The *age* of an application ranks as the second highest correlation among all feature sets, which could be translated as an increased vulnerability discovery window for attackers/users. Those two features are closely related to #CVEs. This could indicate that human factors from both developers and attackers/users should be considered as non-negligible factors in a vulnerability discovery process. From cascaded model, we conclude that the size of a project is also closely related to #CVEs. Small projects (<727KiB in our experiment) often correlate with a smaller #CVEs and vice versa. Despite the correlations, the K-S test shows that none of the normalized features are under the same distribution with #CVEs.

R2: The *DT* and *BT* sets are always ranked with better results. These two feature sets span different metric types. The *DT* set contains *#watches* from popularity metrics, *#contributors* and *#commits* from developer metrics, and the *age* of applications. Thus, this set mostly considers human-related factors and completely ignores the software and security metrics. On the other hand, the *BT* set contains most of the software metrics and two other security metrics features. The predicted results from these feature sets are strongly correlated to #CVEs.

Table VII presents the correlations between the feature sets. Besides the common features, *#watches* and *#forks* are strongly correlated. Based on the fact that both feature sets generate the best prediction results, we could guess that human-related factors play significant roles in the vulnerability discovery process. However, the software metrics and the security metrics provide better interpretation of the vulnerability discovery process for certain applications.

R3: The *BT* model predicts the best #CVEs with the *DT* set, and the overall accuracy is around 77% when the tolerance range is [-5,5]. The correlation between predicted values and the responses is 0.875, which demonstrates that the predicted trend is very similar to that of #CVEs in applications.

B. Use cases

The prediction models are useful in certain contexts where #CVEs for an application is unavailable or unreliable:

I. Enterprises may develop their own internal tools that are not known from the outside world and therefore not listed in any public vulnerability database. They may benefit from comparing their own application to the number of discovered vulnerabilities in open-source applications to accompany a code review. In this case, popularity metrics may be estimated internally at the company or borrowed from known counterpart applications.

II. Also, in the case of an invitation to tender, applicants could be evaluated based on a predicted #CVEs in their base software. This study would be particularly relevant in the military sector where such applications are confidential and thus have no public track record. We conducted a further experiment to evaluate the predictive power of such study by removing the popularity features from our dataset using the *BT* model. The prediction results follow the #CVEs with a correlation of 0.86. We omit the results here to avoid repetitions.
Regarding open- vs. close-source applications, there are indications that at least the vulnerabilities that affect both types of applications are similar. Indeed, Schryen and Kadura [31] conducted an empirical comparison of published vulnerabilities in open-source and closed-source software with multiple applications in various categories. They found that the difference between them is not significant in terms of mean time between disclosures, correlation between age and number of vulnerabilities, and severity levels of said vulnerabilities. Eventually, perhaps we can infer that the similar features are associated with vulnerability discovery in both types of applications.

III. Unify security assessment for applications among various vulnerability databases [32]. Unlike the NVD database, certain vulnerability databases, such as China National Vulnerability Database of Information Security (CNNVD), relies on their own labeling system to register vulnerable applications. In other words, the applications that exist only in the CNNVD would be registered under a disparate system than NVD. Our prediction models provide a possible mapping of #CVEs for such applications from CNNVD to NVD, which further provides a security comparison between those applications to the ones in NVD.

IV. Evaluate the vulnerable open source software projects outside the CVE and NVD. Multiple reasons cause the existence of known vulnerabilities to not have CVE labels in certain types of open source software projects [33]. For example, Snyk's database[8] shows that only 11% of npm package vulnerabilities correspond to a CVE ID. The majority of the vulnerabilities are either not associated with CVE or not listed on NVD. To this extent, our prediction models bridge this gap and provide a possible solution to evaluate these types of projects.

C. Limitations

Guidelines. The accuracy of our prediction models enables us to draw some conclusions about the possible importance of selected features due to the different feature sets that we tested. However, because of the black-box nature of most of the machine learning models we use, those models do not directly provide interpretable patterns. Also, one of our findings related to the statistical models (Table IV) indicates that there might not exist any straightforward correlation between the features and vulnerabilities. Thus, our future work will study the interpretable learning models that could provide guidelines.

Software evolution. The software metrics we consider in this work are collected from the latest version available at the time of collection. It is of interest to note that several such metrics are selected in the *BT* set, which is used to achieve one of the best predictions. This would indicate that current software metrics help predict the number of past discovered vulnerabilities. However, the importance of such metrics in the prediction among other features is unclear. Also, it could be argued that the source code of an application changes over time, and in particular those features may change before and after the vulnerability is fixed. There are several reasons why we do not try to capture features directly from versions affected by CVEs or even older versions:

I. For instance, fixing a buffer overflow vulnerability in C may only require adjusting a buffer size or changing a `strcpy` to `strncpy`, which overall makes little difference in the source code. Therefore, we would expect only a marginal impact on the features, e.g., a handful of lines of code added or removed, few more commits, and a small reduction in the number of flaws identified by FlawFinder.

II. Considering an application is affected by a number of CVEs over the years, which version should we pick to capture our features? It might be possible to combine observations throughout the versions of an application. However, we are not going in this direction in this work since: a) capturing past time-dependent features is non-trivial (e.g., past popularity is not available on GitHub); b) it introduces another dimension to the problem (i.e., versions) that would vary significantly from one application to another and introduce sparsity in the data, which may cause machine learning algorithms to overfit the data; c) a possible "averaged" observation could be obtained for each

[8]https://snyk.io/vuln

TABLE VII: Correlation between *DT* and *BT* set

BT DT	#forks	#contributors	#commits	age	%C	size	#files	#program-files	#comment	#code	L2	L4
#watches	0.95	0.67	0.31	0.12	-0.08	0.08	0.24	0.26	0.17	0.23	0.05	0.03
#contributors	0.73	-	0.44	0.13	-0.08	0.13	0.25	0.28	0.19	0.25	0.05	0.04
#commits	0.33	0.44	-	0.21	-0.05	0.69	0.64	0.57	0.56	0.63	0.19	0.11
age	0.15	0.13	0.21	-	-0.03	0.06	0.13	0.11	0.06	0.10	0.07	0.04

application; however, more work would be needed to evaluate how to properly "flatten" several observed versions into a meaningful feature vector across applications.

By taking the last version, we argue that it somehow captures some history of this application, and in particular it is the result of decisions made to fix the discovered vulnerabilities. Not only does the number of commits capture this evolution, but also the number of contributors would be expected to increase, software metrics and property metrics may also evolve, e.g., code refactoring may impact the number of functions, and the number of comments.

CVEs vs. vulnerabilities. We need to distinguish *actual* and *discovered* vulnerabilities. Our model takes into account #CVEs as they are reported at some point in time. Although this number is the best indication of (past) vulnerabilities in an application, it is nonetheless incomplete as new vulnerabilities may be found later and impact a number of previous versions including the one that we considered. Thus, a given repository may contain an unknown number of previously unidentified vulnerabilities for which some of our metrics could lead our models to overestimate the #CVEs (but not the number of potential vulnerabilities).

Furthermore, #CVEs that affect an application also depends on human factors, i.e., whether the project will receive enough attention from security-minded individuals or organizations to read, review or audit the source code. Our model tries to capture both overall trends of the application's characteristics and human factors.

Zero CVEs. By design, we have not included any repository that is unaffected by any CVE. This allows us to assume a *closed world* of applications that are affected by at least one known vulnerability. Indeed, the number of open-source applications with a non-zero #CVEs is bounded, and such repositories are identifiable. The same does not hold for 0-#CVEs repositories.

Other applications that do not share this property would change our assumption to an *open world*, and more challenges may arise. For example, which proportion of 0-#CVEs applications should we include? There are many GitHub repositories that only host some small scripts/tools that may never be looked at for vulnerabilities despite their popularity. Others may contain non-software data such as (as we found): text-based documents, configuration files, math or proof files, graphics files, game scripting languages, as well as many applications written in exotic languages not used by the general public or not running on common hardware (e.g., Fortran, VHDL, Elixir, Kotlin, SaltStack, NewLisp, QML, AMPL, to name of few of what we have seen). It is not clear whether we should filter out these "noisy" repositories or keep them. Also, we would need to leverage a different list of repositories that include such CVE-free applications. The choice of a list would be critical to avoid biases. To remain in a closed-world assumption, all GitHub repositories would need to be considered, which is arguably difficult to handle.

VI. Related Work

Two major types of vulnerability models have been studied in the literature as a subfield of software security. One focuses on studying the features that correlate with vulnerable components in an application. It is known as the vulnerability prediction model (VPM). The other one focuses on using mathematical models to fit the vulnerability discovery model (VDM) to the historical data, and then to predict the future number of vulnerabilities for one application.

One of the first works in VPM is by Shin and Williams [3], [34]. Those works evaluate the ability of complexity metrics to discriminate between vulnerable and non-vulnerable functions. Zimmermann et al. [35] analyze the possibility of predicting vulnerable components in Windows Vista by using Logistic Regression for five groups of metrics: churn, complexity, coverage, dependency, and organizational. Binary results are evaluated with tenfold cross-validation that yields a precision below 67% and recall below 21%. Meneely and William [36] study the developer-activity metrics and software vulnerabilities. The precision and recall from the Bayesian network predictive model in that study are between 12%–29% and 32%–56%, respectively. Doyle and Walden [37] study the relationships between software metrics and vulnerable components in 14 open source web applications. In that work, the Spearman's rank correlation between the metrics and security resources indicator (SRI) is calculated and obtained based on security scanners. This type of VPMs, which is based on software metrics to pinpoint vulnerable functions/files/components in an application, requires low inspection effort; however, it often suffers from a high false positive rate.

Chowdbury et al. [38] conduct a study to show that complexity, coupling, and cohesion (CCC) related structural metrics are important indicators of vulnerable components. The authors identify 75% of the vulnerability-prone files from 52 Mozilla Firefox releases with a 30% false positive rate. Moshtar et al. [39] propose a set of coupling metrics, which considers the iteration of application modules, thus improving the recall from 60.9% to 87.4% for cross-project vulnerability prediction. Shin and Williams [40] perform a binary classification using Logistic Regression with tenfold cross-validation to analyze the relationship between complexity, code-churn and developer-activity (CCD) metrics and vulnerable components. The authors use 18 complexity metrics, 5 code churn metrics and a fault history metric presented in [2]. The recall and precision from that study are 83% and 11%, respectively.

Other specific applications from VPM are often based on code-specific features. Perl et al. [6] propose VCCFinder to analyze the effects of metadata in the code repositories with code metrics to predict vulnerable commits. The authors study 55 C/C++ GitHub projects including 640 vulnerabilities. By contrast, in our study we leverage 14 times more applications and 10 times more vulnerabilities. The precision of VCCFinder is 60% while the recall is 24%. Younis et al. [41] study the relationship between software metrics and vulnerable functions in existing exploits. In that work, 183 vulnerabilities from NVD for the linux kernel and Apache HTTP server are examined. Stuckman et al. [42] add a code token list to identify vulnerable components. They conclude that the token-based metrics reveal more information than the software metrics in predicting vulnerable components. Walden et al. [43] compare the predictive powers between software metrics and text mining in predicting vulnerable components. Davari et al. [44] propose an automatic vulnerability classification framework based on the features that are extracted from textual reports and the code that fixes vulnerabilities. Li et al. [45] design a deep learning-based vulnerability detection system, Vulnerability Deep Pecker (VulDeePecker), to automatically extract features from vulnerable code fragments from one product and to predict vulnerabilities in other products.

Mathematical VDMs focus on modeling the discovery process of software vulnerabilities by evaluating the number of vulnerabilities with time. The common existing models in the literature are Linear [7], Exponential [8], Alhazmi Malaiya Logistic (AML) [46], and the effort-based model. Woo et al. [47] study both time and effort-based vulnerability discovery models based on Apache and IIS. Massacci et al. [48] conduct an empirical study related to VDMs, and conclude that the simplest linear model is an appropriate choice in terms of quality and predictability for the first 6-12 months after releasing the data. Further than predicting the accumulated number of vulnerabilities, Johnson et al. [49] propose a model called the time between vulnerability disclosure (TBVD) to evaluate the likelihood of finding a zero-day vulnerability within a given time-frame. However, all of these models are specific to one application and normally a large data history is needed to obtain a better-fitted model.

VII. Conclusion and Future work

In this paper, we investigated the possible relationships between software features and the number of vulnerabilities. To the best of our knowledge, this is currently the most comprehensive study to date, as it contains 780 applications, including 6,498 vulnerabilities. We find that machine learning models could help to predict the number of CVEs in an application. In our single-model study, the trends of the predictions of two models are similar to that of the real numbers. These results could be used as a relative security comparison among applications. However, these predicted results are not accurate enough to serve as an absolute security evaluation for an application. In our study, the best overall accuracy we achieve is 77% ± 5 CVEs. In the end, the accuracy could be improved by using cascaded models. We interpret that the small and large applications should be treated with different prediction models to improve the accuracy.

The possible future directions for this work are as follows:

- First, the features we gathered are limited to GitHub projects only. Our future work will expand this study to other open source software projects websites, e.g., SourceForge.
- Second, semantic code related features, such as *Code Gadget* (a number of program statements, which are semantically related to each other [45]), have not been taken into consideration in this work. A future direction is to add more code-related features to complete the feature sets and to repeat the experiments to observe the evolution of the predictive power.
- Third, the machine learning methods we applied are mostly blackbox structures, and we plan to apply other methods whose results might be interpretable.

Acknowledgments

We are grateful to the first author's thesis committee members and the anonymous TIFS reviewers for their comments and suggestions. The first and third authors are partially supported by the Natural Sciences and Engineering Research Council of Canada under a Discovery Grant. The second author is supported in part by a Vanier Canada Graduate Scholarship (CGS).

References

[1] F. Akiyama, "An example of software system debugging," in *IFIP Congress (1)*, vol. 71, pp. 353–359, 1971.

[2] 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.

[3] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in *ACM Workshop on Quality of Protection (QoP'08)*, 2008.

[4] A. E. Hassan, "Predicting faults using the complexity of code changes," in *International Conference on Software Engineering (ICSE'09)*, 2009.

[5] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener, "The relationship between evolutionary coupling and defects in large industrial software," *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[6] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Computer and Communications Security (CCS'15)*, 2015.

[7] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *International Conference on Software Engineering (ICSE'84)*, 1984.

[8] E. Rescorla, "Is finding security holes a good idea?," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 14–19, 2005.

[9] "National Vulnerability Database." https://nvd.nist.gov/, 2017.

[10] S. Rahimi and M. Zargham, "Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 395–407, 2013.

[11] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[12] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[13] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship," *Journal of Software Engineering & Applications*, vol. 2, no. 3, pp. 137–143, 2009.

[14] A. Danial, "Count lines of code (cloc)." https://github.com/AlDanial/cloc, 2017.

[15] D. Wheeler, "Flawfinder," *https://dwheeler.com/flawfinder/*, 2017.

[16] P. Manadhata and J. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, pp. 371–386, May 2011.

[17] J. Pohjalainen, O. Räsänen, and S. Kadioglu, "Feature selection methods and their combinations in high-dimensional classification of speaker likability, intelligibility and personality traits," *Computer Speech & Language*, vol. 29, no. 1, pp. 145–171, 2015.

[18] J. Reunanen, "Overfitting in making comparisons between variable selection methods," *Journal of Machine Learning Research*, vol. 3, no. Mar, pp. 1371–1382, 2003.

[19] M. A. Hall, *Correlation-based feature selection for machine learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, Apr. 1999.

[20] R. Battiti, "Using mutual information for selecting features in supervised neural net learning," *IEEE Transactions on Neural Networks*, vol. 5, no. 4, pp. 537–550, 1994.

[21] K. Kira and L. A. Rendell, "The feature selection problem: Traditional methods and a new algorithm," in *National Conference on Artificial Intelligence*, vol. 2, pp. 129–134, 1992.

[22] J. Mingers, "An empirical comparison of selection measures for decision-tree induction," *Machine Learning*, vol. 3, no. 4, pp. 319–342, 1989.

[23] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[24] C. W. Dunnett, "A multiple comparison procedure for comparing several treatments with a control," *Journal of the American Statistical Association*, vol. 50, no. 272, pp. 1096–1121, 1955.

[25] C. Vercellis, *Business intelligence: data mining and optimization for decision making*. John Wiley & Sons, 2011.

[26] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," *Statistics Surveys*, vol. 4, pp. 40–79, 2010.

[27] MathWorks.com, "Divide data for optimal neural network training." https://www.mathworks.com/help/nnet/ug/divide-data-for-optimal-neural-network-training.html.

[28] J. W. Sammon, "A nonlinear mapping for data structure analysis," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 401–409, 1969.

[29] G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding," in *Advances in Neural Information Processing Systems*, pp. 857–864, 2003.

[30] L. van der Maaten and G. Hinton, "User's guide for t-SNE software," 2008. https://lvdmaaten.github.io/tsne/User_guide.pdf.

[31] G. Schryen and R. Kadura, "Open source vs. closed source software: towards measuring security," in *Symposium on Applied Computing (SAC'09)*, 2009.

[32] Forum of Incident Response and Security Teams, "Vulnerability database catalog." https://www.first.org/global/sigs/vrdx/vdb-catalog.

[33] G. Podjarny, *Securing Open Source Libraries*. O'Reilly, 2017.

[34] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Empirical Software Engineering and Measurement (ESEM'08)*, 2008.

[35] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *International Conference on Software Testing, Verification and Validation (ICST'10)*, 2010.

[36] A. Meneely and L. Williams, "Strengthening the empirical analysis of the relationship between Linus' law and software security," in *International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*, 2010.

[37] M. Doyle and J. Walden, "An empirical study of the evolution of PHP web application security," in *International Workshop on Security Measurements and Metrics (Metrisec)*, 2011.

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

[39] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Symposium on Applied Computing (SAC'16)*, 2016.

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

[41] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit," in *Conference on Data and Application Security and Privacy (CODASPY'16)*, 2016.

[43] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *International Symposium on Software Reliability Engineering (ISSRE'14)*, 2014.

[42] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Transactions on Reliability*, vol. 66, no. 1, pp. 17–37, 2017.

[44] M. Davari, M. Zulkernine, and F. Jaafar, "An automatic software vulnerability classification framework," in *International Conference on Software Security and Assurance (ICSSA'17)*, pp. 44–49, IEEE, 2017.

[45] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Network and Distributed System Security (NDSS'18)*, 2018.

[46] O. H. Alhazmi and Y. K. Malaiya, "Prediction capabilities of vulnerability discovery models," in *Annual Reliability and Maintainability Symposium (RAMS'06)*, 2006.

[47] S. Woo, H. Joh, O. H. Alhazmi, and Y. K. Malaiya, "Modeling vulnerability discovery process in Apache and IIS HTTP servers," *Computers & Security*, vol. 30, no. 1, pp. 50–62, 2011.

[48] F. Massacci and V. H. Nguyen, "An empirical methodology to evaluate vulnerability discovery models," *IEEE Transactions on Software Engineering*, vol. 40, no. 12, pp. 1147–1162, 2014.

[49] P. Johnson, D. Gorton, R. Lagerström, and M. Ekstedt, "Time between vulnerability disclosures: A measure of software product vulnerability," *Computers & Security*, vol. 62, pp. 278–295, 2016.

Mengyuan Zhang is an Experienced Researcher at Ericsson Research, Montreal, QC, Canada. She received her Ph.D. in Information and Systems Engineering from Concordia University in Montreal. Her research interests include security metrics, attack surface and cloud computing security. She has published several research papers and book chapters on the aforementioned topics.

Xavier de Carné de Carnavalet is a Ph.D. candidate in Information and Systems Engineering at Concordia University, Montreal, QC, Canada. He received in 2014 a Dipl.-Ing. from École Supérieure d'Informatique, Électronique et Automatique, Paris, France, and an M.A.Sc. in Information Systems Security from Concordia University. His research interests are: privacy, passwords and authentication, TLS interception, trusted computing, reverse-engineering, and machine learning applications to information systems security.

Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Quebec, Canada. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. He holds a M.E. from Shanghai Jiao Tong University and a B.E. from Shenyang Aerospace University in China. His research interests include cloud computing security, network security metrics, software security, and privacy. He has co-authored five books, two patents, and over 100 refereed conference and journal articles at top journals/conferences, such as TOPS, TIFS, TDSC, TMC, JCS, S&P, CCS, NDSS, ESORICS, PETS, ICDT, etc. He is serving as an associate editor for IEEE Transactions on Dependable and Secure Computing (TDSC) and he has served as the program (co)-chair of seven international conferences and the technical program committee member of over 100 international conferences.

Ahmed Ragab received the Ph.D. in Industrial Engineering from Polytechnique Montréal, Canada, in 2014. He has received the M.Sc. in Control Engineering in 2007 and the B.Sc. in Electronic Engineering in 2003, from the Faculty of Electronic Engineering, Menouf, Egypt. His research interests are: Control Engineering, Machine Learning, Operations Research, Discrete Event Systems, Maintenance and Reliability Engineering, Fault Diagnosis and Prognosis and Decision Support Systems.