Abstract—Software vulnerability has been the root cause of many high profile security incidents. Consequently, whether a new software will have many vulnerabilities in the future may become an important factor to consider, when administrators need to decide whether to deploy a newly released software, to choose between alternative software, or even to update a software to its newer versions (software upgrade is known to potentially introduce vulnerabilities). Although vulnerability scanners and public vulnerability databases may help identify known vulnerabilities in existing software, they are less effective for finding zero day vulnerabilities in newly released software or new versions of a software. On the other hand, existing research on vulnerability discovery models shows that, while the existence of vulnerabilities in a software may be linked to certain features, e.g., size or complexity, of that software, such a correlation is usually not straightforward or reliable enough for predicting vulnerabilities for one particular software, 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 software and the abundance of vulnerabilities. Unlike existing works, which typically focus on one particular software resulting in limited successes, we focus on the more realistic issue of predicting the relative likelihood of vulnerabilities among different software. To the best of our knowledge, this is the most comprehensive study on 780 real world software applications involving 6,498 vulnerabilities. We apply seven feature selection methods on nine feature subsets selected among 34 collected features, which are then fed into six types of learning model, producing 523 estimations. The predictive power has been evaluated using four different performance measures. Our study shows that some human factors help explain the number of discovered vulnerabilities.

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

I. INTRODUCTION

SOFTWARE vulnerabilities are usually considered a key threat to mission critical networks, such as those used in power plants, governmental or military organizations, and cloud data centers. A practical question often faced by security administrators is: Will this new software have many vulnerabilities in the future? While existing vulnerability scanners (e.g., Nessus [1]) and public vulnerability databases (e.g., NVD [2]) make it easy to identify known vulnerabilities in well-known software, these become less effective when it comes to finding unknown vulnerabilities, in newly released software or new versions of a software, since these tools mainly depend on prior knowledge about vulnerabilities or software.

On the other hand, existing research on vulnerability discovery models (VDMs) shows that vulnerabilities in a software may be linked to certain features, e.g., size or complexity, of that software (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 a software based on its features? However, as shown in most existing works on VDMs, including those that focus on predicting vulnerable components inside one software [3], [4], [5], [6], [7], [8] and those aim to establish mathematic models based on the historical vulnerability data of one software [9], [10], the correlation between vulnerabilities and the features of a software is usually not straightforward or reliable enough for such a prediction, e.g., the number of vulnerabilities in a software is not necessarily 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 software may depend on multiple factors in a complex way, which is still largely unknown.

In this paper, unlike existing works that are typically study on function/file/component level, we focus on the more realistic issue of predicting the relative likelihood of vulnerabilities among different software, i.e., which software is more likely to contain vulnerabilities. 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 a software and a rich collection of features. 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 and correlation.

A. Overview

Figure 1 provides an overview of our study. Specifically,

- First, we obtain a large-scale dataset of open-source applications from GitHub each of which involves at least one vulnerability listed in NVD [2]. We address various challenges in the data collection, e.g., automatically identifying GitHub repositories, obtaining the accurate
A common belief is that the complexity of an application is the main cause of vulnerabilities [5]. However, the reality is more complex since many other factors may come into play. For example, in our GitHub dataset, project opencms-core,\textsuperscript{1} 780,638 lines of code, which has lower complexity than project SiberianCMS,\textsuperscript{2} 1,863,840 lines of code, actually turns out to have a higher number of vulnerabilities. Upon closer examination, we find that project opencms-core 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 are not likely to yield sufficient prediction power. Therefore, we set up three research questions and summarize our main findings as follows.

- **R1**: There exists one feature that is significantly correlated with the number of vulnerabilities in different applications.

  **Our Finding**: In literature, correlation is interpreted as follows: correlation less than 0.3 value of correlation coefficient means weak correlation, 0.3 to 0.5 means medium correlation, and greater than 0.5 means strong correlation [5]. If we follow the same interpretation, only two features fall into medium correlation, the number of commits from developer metrics and age from software property metric. The rest of the features will be considered as weak correlation to #CVEs. In the later discriminative test, all the features are rejected in K-S test with small $p$-value.

  The conclusion we can have from this research question is that: instead of complexity, human factors share higher correlation with #CVEs, the number of commits (from developer metrics) and age (from software property metrics, but also indicates the attack windows for an application) feature share medium positive correlation with #CVEs. However, even the best correlated features share different distribution with #CVEs, which means none of the features are not in the same distribution as #CVEs.

- **R2**: There exists a combination of features that is significantly correlated with the number of vulnerabilities in different applications.

  **Our Finding**: In our experiments, the feature sets selected based on the embedded methods with Decision tree (DT) and Boosted tree (BT) algorithms have relatively good performance metrics and accuracy, as detailed in Section IV. The correlation from DT feature set is 0.875 and the correlation from BT feature set is 0.845, which both are considered as strong correlated to #CVEs.

\textsuperscript{1}https://github.com/alkacon/opencms-core

\textsuperscript{2}https://github.com/Xtraball/SiberianCMS
• R3: Machine learning methods applied to those features can effectively predict the number of vulnerabilities in different applications.

Our Finding: BT classifier yields the best results with the DT feature set, and the overall accuracy is around 77% when the tolerance range is [-5,5], as detailed in Section IV, which could provide a rough indicator about relative abundance of vulnerabilities. In cascaded Model analysis, we discover that the size of a project could indicate.

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 the learning based prediction models. Section IV analyzes the software vulnerability prediction models. Section VI reviews related work, and finally Section VII discusses limitations and gives concluding remarks.

II. DATA COLLECTION AND FEATURE EXTRACTION

In this section, we describe the collection of our dataset of open source applications that are affected by at least one vulnerability listed in NVD [2], along with features extracted from both the source code and metadata provided by GitHub.

Overview. We consider GitHub as an open-source platform as it will allow us to easily locate and retrieve the source code of possibly many applications for our study. Next, we investigate the last 10 years of CVE vulnerabilities (from January 2008 to October 2017, included) in search for applications found on GitHub. In total, we consider 67,294 CVEs, which distribution by year 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, we need to identify whether they belong to the official application repository affected by the CVE. The label reference_type is provided on each URL as a hint. However, we found that it is often unreliable, i.e., an official application repository could be labeled as VENDOR_ADVISORY, PATCH or UNKNOWN, without clear rules.

At this stage, we take 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 indicate a short-lived content related to the CVE or simply 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 to the fork only or also the forked application. We also verify that the given URL is not a simple advisory or PoC by searching for keywords in the repository’s name and description. In parallel, we keep track of all GitHub repositories listed as references to help resolve certain conflicts at a next stage.

Another challenge is the change of a repository owner or application name, 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, 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 a very unreliable indicator of the true number of 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 the application’s project did not exist on GitHub prior to a certain date. Worse, a GitHub reference may be missing in several CVEs that affect the same repository, e.g., tomlhughes/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 counting the CVEs where either the URL of the repository or the affected application is listed. For example, tomlhughes/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

\[\text{CVE}\backslash b\mid \text{PoCs}\backslash b\mid \text{PoCs}\backslash b\mid \text{SQL}\backslash b\mid \text{SQL}\backslash b\mid \text{XSS}\backslash b\mid \text{XSS}\backslash b\mid \text{BB}\mid \text{BB}\mid \text{ug}\backslash a\backslash b\].

\[\text{We match the name of the repository with the following regular expressions (given in Python syntax):}\]

\[\begin{aligned}
\text{\text{CVE}}\backslash \text{b}\mid \text{PoCs}\backslash b\mid \text{PoCs}\backslash b\mid \text{SQL}\backslash b\mid \text{SQL}\backslash b\mid \text{XSS}\backslash b\mid \text{XSS}\backslash b\mid \text{BB}\mid \text{BB}\mid \text{ug}\backslash a\mid b\end{aligned}\]
product may be referred in various ways, e.g., best-practical/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 to 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 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 to help in manual labeling of repositories and mapping to affected products. It is based on a heuristic scoring system to assist the human expert performing the rest of labeling by suggesting the most probable corresponding products for a given repository, and rules to automatically attribute products to repositories and count CVEs when ambiguities are easily resolved.

1) Scoring system: After we scan 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 owner/name and the affected product’s organization/name. We define the similarity as a scoring system based on heuristics we developed while performing manual labeling on a chunk of the dataset.

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) between both represent less that 10% of the longest string. This lose comparison encompasses small variations and gives 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 subparts. 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 the 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 Figure 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 dash, underscore and digit 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., phpbh: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 points 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 the threshold higher 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 the number of CVE. 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 less than 5 stars, no fork, and consists of 15 files or less. 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/noncedispect), manually labeled mirrors/read-only repositories (e.g., LibreOffice/core), or empty repositories. Finally, 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 number of CVE and further consider 788 repositories.

D. Example

Figure 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 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 bestpractical: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. Our tool would properly account for such an overlap otherwise. 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

After cloning the 788 GitHub repositories previously shortlisted (however, eight repositories systematically crash the tools we use to extract certain features. In the end we only collected features for 780 repositories), we also proceed to retrieve certain metadata from GitHub 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.

1) Popularity Metrics: The popularity metrics translate 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. The number of forks is the number of the copies of the projects made on GitHub by other users. Other contributors could work on a fork, make additions or fix bugs. They could later send a pull request to the original owner to include their modifications into the original code base. Therefore, the number of the forks reveals the popularity of the repository among other active developers. The number of stars and watches on the project show the attention of the repository among interested GitHub
TABLE II: The Overview of GitHub Applications Features

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Features</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Popularity</td>
<td>Number of stars</td>
<td>#stars</td>
</tr>
<tr>
<td></td>
<td>Number of watches</td>
<td>#watches</td>
</tr>
<tr>
<td></td>
<td>Number of forks</td>
<td>#forks</td>
</tr>
<tr>
<td>Developer</td>
<td>Number of contributors</td>
<td>#contributors</td>
</tr>
<tr>
<td></td>
<td>Number of commits</td>
<td>#commits</td>
</tr>
<tr>
<td>Software</td>
<td>Age</td>
<td>age</td>
</tr>
<tr>
<td></td>
<td>Number of labels</td>
<td>#labels</td>
</tr>
<tr>
<td></td>
<td>Language distribution</td>
<td>%+language, e.g., %Java</td>
</tr>
<tr>
<td>Software</td>
<td>Size</td>
<td>size</td>
</tr>
<tr>
<td></td>
<td>Number of files</td>
<td>#files</td>
</tr>
<tr>
<td></td>
<td>Number of program files</td>
<td>#program-files</td>
</tr>
<tr>
<td></td>
<td>Number of blank lines</td>
<td>#blank</td>
</tr>
<tr>
<td></td>
<td>Number of comment lines</td>
<td>#comment</td>
</tr>
<tr>
<td></td>
<td>Number of code lines</td>
<td>#code</td>
</tr>
<tr>
<td></td>
<td>Number of lines of C/C++</td>
<td>c-sloc</td>
</tr>
<tr>
<td>Security</td>
<td>Number of issues</td>
<td>#issues</td>
</tr>
<tr>
<td></td>
<td>Number of functions</td>
<td>#functions</td>
</tr>
<tr>
<td></td>
<td>Flawfinder risk levels</td>
<td>hits, L1-4.5</td>
</tr>
</tbody>
</table>
of functions, which could be considered as an upper bound of attack surface, to replace attack surface as another potential attack likelihood metric for an application. We first use c-tag\(^4\) to index all functions. Then we sum up all the functions to get the total number of functions in a project.

The number of issues inside one project contains the software bugs that reported from users and also the tasks for project maintainers. We consider that some of the issues are related to security issues, e.g., issue #6599 in openssl, a bug related to accept invalid certificate versions\(^5\), which could lead to security vulnerabilities. Therefore, we obtain the number of open issues that have been released to each project as the existing attack likelihood.

III. THE FEATURE SELECTION

In this section we use machine learning technologies to evaluate the effectiveness of the features we collected to the defined target (the number of CVE vulnerabilities) in GitHub dataset. Before we continue, 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.

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 [16] 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 classification performance, lower the computational costs, simplify the classifiers and provides better understanding in the classification problems [17]. Three types of feature selection methods, namely, the filter methods, the wrapper methods and the embedded methods, are applied in our study to achieve feature subsets.

Filter methods evaluate the score of each feature according to certain criteria and the experts then choose the subsets based on the scores, e.g., correlation-based feature selection (CFS), and mutual information (MI). However, filter methods only consider the relationship between the pairs of features; the relationships between multiple features are ignored in this type of feature selection methodology.

Wrapper methods involve the learning algorithms to evaluate the relevance of the feature sets, e.g. k-nearest neighbors algorithm (k-NN), which classifies the object based on the k-nearest neighbors. 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 computation time in searching for the best feature subsets from the possible permutations of the feature subsets grows exponentially with the number of features; heuristic algorithms, such as sequential forward selection (SFS), are proposed to tackle this search problem. SFS starts from an empty set and adds the feature in order to obtain the maximal score in the iteration to the feature set.

Embedded methods build the feature selection process inside the learning algorithm, e.g., decision trees [18], random forests [19]. The feature selection process is using the entire dataset as the input to generate the feature.

Seven feature selection methods from filter methods, wrapper methods, and embedded methods, are applied to our feature sets to select the best feature subset to reduce the errors, which would be generated by correlated or noise features.

1) Filter Methods: In filter method, we choose correlation-based feature selection (CFS) [20], mutual information (MI) [21] and ReliefF [22], due to the fact that these methods are wildly used in the literatures.
## Wrapper Methods

Table III demonstrates the feature selection results from all the methods we mentioned in the previous section. In the latter section, the classifiers would be built on the full feature sets. In the experiment, we use 200, 500 and 1000 trees in our study. The results are almost identical with the change of number of trees. In this study, we select features from the feature sets.

### 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. CFS method selects the features that highly correlate with the response and do not correlate with other features. Figure 3(a) is the feature selection scores from CFS. In this study, we first set 0.01 as the threshold for the CFS method, which means the weight lower than 0.01 is preselected in this step. Then we joint the results from CFS with the Table IV correlation column to build the feature set.

### b) Mutual Information

Mutual information (MI) measures the mutual dependence between the response and the predictors. Same as the correlation, MI only produce pairwise results. Contrasted with the correlation, MI captures the non-linear dependency through the joint probabilities. In this study, we output the MI score as the importance weight for features. Figure 3(b) demonstrates the importance for features with the features as x-axis and MI score as y-axis. The threshold to select the feature subset is 0.15 in this study.

### c) ReliefF

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

### 2) Wrapper Methods

Multiple heuristic algorithms are available to build the best set in wrapper methods; in this study, we apply the SFS and sequential forward floating selection (SFFS); both are the family of greedy search algorithms, with \(k\)-NN classifier to select the feature set. Since SFFS is an extension of the SFS, in our study, both algorithms give the same output feature set. The SFS algorithm starts with an empty set and adds one feature, which gives maximum accuracy within each iteration. Six features are selected in the wrapper methods.

### 3) Embedded Methods

Three are embedded methods that are applied to our dataset: Decision tree (DT), Boosted tree (BT), and Random forest (RF). Figure 3(d) demonstrates the feature selection results from the DT algorithm; the results are plotted as the importance bars. We select 0.05 as our threshold to select features from the feature sets.

Then we apply the boosted tree with 200, 500 and 1000 trees in our study. The results are almost identical with the different number of trees, therefore, we only show the result from 200 trees as an example here. Figure 3(e) shows the feature selection result from BT. In this study, we set 0.018 as the threshold to select the feature subset.

Figure 3(f) shows the feature selection from RF 1000 trees. In the experiment, we use 200 trees, 500 trees and 1000 trees to test the feature importance weight, and the results for the important weight only have neglected difference regarding with the change of number of trees. In this study, we select the feature subset based on the result from 1000 trees with 0.1 as the threshold.

Table III demonstrates the feature selection results from all the methods we mentioned in the previous section. In the latter section, the classifiers would be built on the full feature sets and the selected feature sets from various feature selection
IV. THE ANALYSIS

In this section, we first apply two statistical methods to evaluate the discriminative power of features. Then, the prediction powers of learning based models are analyzed in Section IV-B. Finally, we discuss Research Questions in Section V.

A. Statistical Analysis

In this section, we apply two statistical methods to evaluate our features against the number of CVE vulnerabilities. First, we use Min-Max standardization to normalize the value of features to obtain bounded results for our dataset.

\[ z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \]  

(1)

where \( x = (x_1, \ldots, x_n) \) are the original data set and \( z_i \) is the \( i^{th} \) normalized data point corresponding to the original data set. This function maps all the features into a same range \((0,1)\).

The first statistical analysis is Pearson coefficient, which illustrates the linear relationships between response and predictor, e.g., in our case, is #CVEs and features. The output result for the correlation coefficient is between -1 to 1, which corresponds to fully opposite, or fully linear correlations. The value 0 means that two sets of the data are not correlated. The second method is two-sample Kolmogorov-Smirnov test (K-S test),\(^6\) which returns a decision, and \( p\)-value, to demonstrate whether the response and the predictor are from the same continuous distribution. Bonferroni correction is used to deal with multiple hypothesis testing, which means our stricter significance level is 0.00029 (corresponding to a non-corrected \( p \leq 0.01 \) for each test).

The top three correlations are the number of commits, age and the number of functions corresponding to the developer metric, software property metric and security metric. According to the interpretation in literature [5], correlation is considered as weak correlation when the value is less than 0.3, our study demonstrates 32/34 features as weak correlation to #CVEs. Only #commits and age are medium correlation (correlation value in 0.3 to 0.5 is considered as medium correlation). However, in K-S test, all the features are rejected due to the low \( p\)-value, which means no feature follows same distribution as #CVEs.

As the conclusion for R1, #commits is the best feature that correlates to #CVEs, which still under medium correlation not strong. Overall, software property metrics have the lowest and unstable correlation, e.g. some languages share weak negative correlation, with #CVEs, ; features in software metrics correlates with #CVEs around similar level, all of them show weak positive correlation. Security metrics and popularity metrics are also showing weak correlations. However, all the features are under different distribution with #CVEs

B. Learning Based Analysis

\(^6\)This test calculates the distance between the distribution of two samples; the null hypothesis is that the two samples are from the same distribution.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Correlation</th>
<th>( p)-value</th>
<th>K-S test</th>
</tr>
</thead>
<tbody>
<tr>
<td>#stars</td>
<td>0.0816</td>
<td>9.11E-67</td>
<td>Reject</td>
</tr>
<tr>
<td>#watches</td>
<td>0.1420</td>
<td>1.72E-71</td>
<td>Reject</td>
</tr>
<tr>
<td>#forks</td>
<td>0.1425</td>
<td>5.38E-66</td>
<td>Reject</td>
</tr>
<tr>
<td>#contributors</td>
<td>0.1307</td>
<td>1.02E-68</td>
<td>Reject</td>
</tr>
<tr>
<td>#commits</td>
<td>0.4360</td>
<td>1.67E-69</td>
<td>Reject</td>
</tr>
</tbody>
</table>

TABLE IV: Results of the Statistical Analysis for GitHub Dataset based on Spearman’s rank correlation coefficient and K-S test. K-S test significant if \( p\)-value \( \leq 0.00029 \)

1) Experiment Evaluation Methods and Results: Unlike some existing work that 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 which responses are numeric values that correspond to a number of CVE per application. We conducted this experiment using six regression models with various parameters on various sets of features, totaling 523 prediction results.

Data Preparation. As Flawfinder only support C/C++ projects (240/780 projects contain at least 50% of C/C++, and up to 302 that contain any proportion of C/C++ in our dataset), we assign the value -1 to all Flawfinder-related features (expectedly positive integers) for projects with zero amount of C/C++. As Flawfinder only support C/C++ projects according to the GitHub repository’s metadata [23].

Cross-validation. Each of the training and prediction experiments has been conducted using ten-fold cross-validation. The same randomized folds are used with linear, NN, and RF models as they do not internally support cross-validation in MATLAB (i.e., we perform 10 training-prediction rounds based on the randomly pre-separated folds, and average the results ourselves), while folds for BT, DT, and SVM models are selected internally.

For the fitting NN model, a validation set is also required to avoid overfitting [24]. 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% [24], but match more closely the proportions of a classic ten-fold cross-validation.

**Performance metrics.** The performance of the models are evaluated through mean-squared error (MSE)/root mean square (RMS), a wildly used function to emphasize large error; 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 gives more emphasis on errors for small number of CVE. Usually, a lower MSE/RMS, MAE, and MAPE correspond to a better performance of a model/feature set. In addition, correction is used to compare the trend between responses and predictors.

**Boosted Tree.** The first classifier we implemented in our experiment is Boosted Tree, BT and BT-opt (BT-opt chooses to calculate parameters with the inbuilt algorithms). Table V demonstrates the evaluation results from MSE, RMS, MAE and MAPE. It is normal that the lowest values of for evaluation methods are coming from different experiment setup. for example, the prediction results from BT-opt with all features is associating with lowest MAPE in this experiment set, however, MSE/RMS and MAE are higher than BT with DT feature set. The main reason for this observation is that evaluation methods are more sensitive to capture errors in certain type of data. For example, MAPE is sensitive to the errors from small values inside one dataset while MSE/RMS captures the existence of large error inside the prediction.

**Decision Tree for Regression.** The five other classifiers we have implemented in our experiment are Decision Tree (DTTr), Logistic Regression (LR), Artificial Neural Network (ANN), Random Forest (RF and SVM). Table VI illustrates the best evaluation results that generated from each classifiers and the corresponding feature sets. DT feature set, which only contains 4 features from three categories, provides the best evaluation results for 5 classifiers. To generalize our diversified dataset, DT feature set contains the most generic features, thus it discards all the application code specific features. This result indicates that the general comparison among applications could be simply generated from developer, popularity and the existing time of the applications. BT feature set, which involves more software and security metrics receive similar overall performance with DT feature set in our study. The prediction result from DTTr with BT feature set is presented in Figure 4(b).

**Linear Regression.** In LR classifier, DT feature set has the best MSE and RMS; SFS feature set has the best MAE and MAPE. The correlation is 0.451 and 0.434 from the predictions, therefore, we only demonstrate the best result from DT feature set in Table VI and show the predicted results and the accuracy in Figure 4(c) and Figure 5(c).

**Neural Networks.** Two Neural Network (NN) classifiers, function fitting NN and generalized regression NN, are applied to our dataset. To obtain the relative better result from both classifiers, we choose 36 hidden layers (from 1 to 36) to get the best MSE, RMSE, MAE and MAPE for different feature sets. BT feature set with hidden layer = 4, associates with the best MSE and RMS and DT feature set with hidden layer = 6, associates with the better MAE and MAPE. Since the correlation is 0.533 for BT feature set, which is higher than the correlation from DT feature set, 0.406, we demonstrate the best result from BT features set in Table VI and show the predicted results and the accuracy in Figure 4(d) and Figure 5(d).

**Results.** The best evaluation results from RF classifier has been chosen from 5 different tree parameter, tree = 10, 100, 200, 500 and 1000. The best result is from DT feature set with tree = 500, which is presented in Table VI and Figure 4(e) and Figure 5(e).
by using the CFS feature set in our entire experiments. We generate the predicted results corresponding to the accuracy in Figure 4(f) and Figure 5(f).

2) Experiment Results Interpretation: To better understand the prediction results, Figure 4 demonstrate the predicted value from the best classifier and feature set combination. In our dataset, multiple observations correspond to the same number of CVEs, we group the observations based on the number of CVEs. X-axeses in the graphs represent the number of unique CVEs from our dataset. The red squares in the upper graph plot the real CVEs for each unique CVE, which are the same as the labels on the x-axis. The green crosses are the average predicted CVEs for each unique CVE and the blue pluses are the prediction CVEs for each observation.

The predicted results are ordered by the relative percentage error ($\frac{\text{average predicted CVEs} - \text{real CVEs}}{\text{real CVEs}}$), for example in Figure 4(a) 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 classifier with DT feature set. The correlation value between real CVE(s) and the predicted CVE(s) is 0.875 in this experiment. The higher correlation means that the trend of the predicted CVE(s) follows the similar trend with the real CVE(s).

Results and Implications for Prediction Results from Classifiers: According the predicted results from six different classifiers, first we can see that the predictions for each response vary among classifiers. For example, the best indicated project (CVEs = 73) in Figure 4(a) is classified as one of the largest errors in Figure 4(d). The majority of projects could be predicted close to original CVEs within certain tolerant range in different classifiers, which will be demonstrated in Figure 5. In general, the some projects with large number of CVEs are well predicted from BT and DT, the projects related with small number of CVEs are better predicted in SVM and the inter medium results are better in LR, ANN and RT.

The correlation between predicted CVEs and real CVEs, which relates to the predictive power of a combination of feature sets, is improved significantly compare to the correlations in Table IV. This means DT and BT feature set captures important factors in software vulnerability discovery process.

Some projects have never been classified close to their real CVEs in the entire experiment. Project tcpdump, which associates with 140 number of CVEs, has never been predicted accurately. We observe that this project got 133/140 vulnerabilities released during 2017 in two different patches. These unexpected release lifted the number of CVEs for this project, which leads to a significant difference between predicted number of CVEs and real number of CVEs in our model. The best predicted value is CVEs = 7 for this particular project from all the models, which matches close to the number of CVEs before these unexpected releases. Project ntp, corresponding to 77 number of CVEs in our data set, also never got predicted accurately in the entire experiment. Similarly, We capture the unexpected large number of CVEs release in 2017 for this project (59/77 were reported in 2017). Our model could not capture the unexpected release for the vulnerabilities due to human/automation results, such as research project, automation testing results, etc.

Relative percentage error is exaggerated within the small number observations since the small difference already generates a large percentage error. For example, the relative percentage is 100 when the predicted number is 2 and the real number is 1. We study relative range of tolerance to understand the predictive power in small number of responses for different classifiers. The x-axis of Figure 5 shows the ranges of tolerance, e.g., [-1,1] means we accept the prediction result with 1 error. 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: Figure 5 demonstrates the predictive power for small number of CVEs for each classifiers based on aforementioned feature set. The red line in each graph shows the overall accuracy for the

---

1https://github.com/the-tcpdump-group/tcpdump
2https://github.com/ntp-project/ntp
entire dataset. Since the majority number of data entry is CVE = 1, the overall accuracy often close to the accuracy of this entry. Although the correlations are similar in BT and DTr classifier, the predictions for small number of CVEs are very different. DTr, which has lower number of MAPE than BT, demonstrate the predictive power for small number of CVEs by having relatively higher accuracy in CVE(s) = 1 and 2. The performance for LR and NN classifiers are worse than the aforementioned ones. These two classifiers predict accurately for inter medium CVE entries, which end up with higher results of the evaluation methods and flatter increase of accuracy. Figure 5(e) demonstrates the advantage of low MAE; the accuracy increases faster then other classifiers. Figure 5(f), SVM with SFS feature set, which has the lowest MAPE, demonstrates the best prediction results for CVE = 1,2,3.

BT and DTr classifiers give the relatively better overall trend prediction, with more accurate predictions for larger number of CVEs. LR and NN classifiers more accurate results for inter medium number of CVEs. RF classifier gives the best average increase of accuracy, thus could be used for the predictions with certain tolerant range. SVM classifier could predict small number of CVEs in the most accurate way. The overall accuracy for the entire dataset reaches 77% with [-5,5] tolerant range.

3) Cascaded Model Analysis: We have noticed classifiers perform better in certain dataset in the previous analysis; to better understand the number of CVEs and the features, we apply unsupervised learning: t-Distributed Stochastic Neighbor Embedding (t-SNE) [25] technique, to cluster our dataset and visualize it into 2D dimension. t-SNE represents the similarities of high-dimensional datapoints as the conditional probabilities that are calculated by the algorithms, e.g., the default algorithm is Euclidean distance algorithm. Compared to the traditional dimensionality reduction techniques, e.g., Principal Components Analysis (PCA) [26], which uses linear techniques, t-SNE performs better in keeping the similar datapoints close together with nonlinear dimensionality reduction techniques, which can not be achieved by a linear mapping. Compared with other nonlinear dimensionality reduction techniques, such as Sammon mapping [27], Stochastic Neighboring Embedding (SNE) [28], t-SNE is capable of capturing both the local structure and the global structure of the datasets.

In t-SNE Matlab implementation [29], 11 distance algorithms can be chosen in the t-SNE function (default distance algorithm is Euclidean distance). Parameter Perplexity controls the effective number of local neighbors at each point (default value is 30). Figure 6 is the visualization of GitHub dataset from four algorithms. In visualization process, we remove the response, the number of CVEs, from the dataset; only predictors are clustered with t-SNE algorithm. t-SNE maps the 34 features into two dimensional coordinate, then we use the number of CVEs to color datapoints. The blue dots are defined as the instances that correspond to instances of applications that are assigned five or more CVEs. The other colors correspond to the exact number of the CVEs. In Figure 6(a), the datasets are visually well divided into five clusters; two of the clusters are majority with lower number of CVEs and three of them are majority with higher CVEs.

We have applied all the distance algorithms with various perplexity values in order to obtain the best visualization results. Figure 6 (a),(b),(c),(d) show the most visually separated visualization results from GitHub dataset.

We design a new cascaded model to first separate original dataset into three datasets according to Figure 6(d), namely cluster 1, 2 and 3. Cluster 1 is the left lower data cluster in Figure 6(d); cluster 2 is the upper data cluster and cluster 3 is the right lower data cluster. Then apply classifiers on clusters to predict the number of CVEs for each project. In our dataset, after apply 10-fold cross validation, the decision tree classifier in Figure 7(b) could separate our dataset into three clusters with 100% accuracy. From the color of the clusters, we could notice cluster 2 mainly contains small number of CVEs (average CVEs = 1.4); cluster 1 is mainly for higher number of 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 is corresponding to the number of CVEs. Small size of projects are always with low number of CVEs, however, some large size of projects are also onl small number of CVEs, which require other features to explain.

We have applied all the classifiers on the separated dataset,
in small size of projects. The accuracy is shown in Figure 8(a), which has 90% of overall accuracy with [-1,1] tolerant range. Cluster 3, which has inter medium number of CVEs, has shown the similar predictive power with the aforementioned studies. We believe more features or selected feature set will increase predictive power for Cluster 3.

V. DISCUSSION

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

A. Research questions

R1: After the statistical analysis of our dataset, we obtain the best correlation, 0.436, between #commits and #CVEs, which shows as medium correlation, in comparison, all the traditional security metrics, the complexity metrics, only show weak correlation to #CVEs. The age of an application ranks as the second highest correlation among the entire feature sets, which could be indicated as vulnerability discovery window for attackers/users. The fact that those two features are closely related to #CVEs indicates that in general vulnerability discovery models, human factors (both developer and attackers/users) should be considered as non negligible factors. From the study of cascaded model, we could conclude that the size of a project also closely related to #CVEs. The smaller size often correlates with smaller number of #CVEs vice versa. Nevertheless, the better correlation from two features, all the features are not under the same distribution with #CVEs.

R2: because DT and BT feature sets are always ranked as the way better results from the experiments. These two feature subsets contains total different features, DT feature set contains #watches from popularity metrics, #contributors and #commits from developer metrics, and age of applications. DT feature set mostly consider human-related factors and ignore the software metrics and the security metrics completely. However, BT feature subset contains most of software metrics and two features from security metrics. Compare to single feature, the predicted results from these feature sets are strongly correlated to #CVEs.

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

B. Use cases

The prediction models are directly useful in certain contexts where the number of CVE for an application is unavailable or unreliable:

1. 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
accompanied by a code review. In this case, popularity metrics may be estimated internally to 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 number of CVEs in their base software. This would be particularly relevant in the military sector where such applications are confidential and thus have no public track record. Popularity metrics may need to be revised or estimated; however, the practical use of our work in such contexts may require additional effort.

C. Limitations

Guidelines. The accuracy of our predictions enables us to conclude on the possible importance of selected features thanks to the different feature sets that we tested. However, due to the black-box nature of most of the machine learning models we use (a common issue in machine learning), those models do not directly provide any patterns that we could interpret. 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, we cannot provide guidelines that could help developers to avoid “bad” behaviors.

Software evolution. The software metrics we consider in this work are collected from the last version available at the time of collection. It could be argued that the source code of an application changes over time, and that the features may change before and after a vulnerability is fixed. There are reasons why we do not try to capture features directly from versions affected by CVEs or otherwise 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 could expect only a marginal impact on our 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 affected by a number of CVEs over the years, which version would we pick to capture our features? It could be possible to combine observations throughout the versions of an application; however, we are not going in this direction in our 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 our data, i.e., the dataset may be too small against the number of dimensions in the problem, and cause machine learning algorithms to overfit the data; c) a possible “averaged” observation could be obtained for each 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 the number of commits captures this evolution, but the number of contributors could be expected to increase, software metrics and property metrics may also evolve, e.g., code refactoring may impact the number of functions, and amount of comments.

CVEs vs. vulnerabilities. We need to distinguish actual and discovered vulnerabilities. Our model takes into account the number of 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 that impact a number of previous versions including the one that we consider. 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, the number of CVEs that affect an application also depends on human factors, i.e., whether the project will receive enough attention for 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.

No CVEs. By design, we have not included any repository that is unaffected by any CVE. This design 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 cannot be said of 0-#CVEs repositories.

Including 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 text-based documents, configuration files, math or proof files, graphics files, game scripting languages, as well as many applications written in exotic languages that are not to be used by the general public or 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 close world scenario, all GitHub repositories would need to be considered, which is arguably difficult to handle.

VI. Related Work

Two major vulnerability models have been studied in the literature as a subfield of software security; one focuses on studying the features that correlate with the vulnerable components in software application, also is known as vulnerability
prediction model (VPM); the other one focuses on using mathematic models to fit the vulnerability discover model (VDM) with the history data then to predict future number of vulnerability for one application.

One of the first works in VPM is from Shin and Williams [5], [30] in which authors evaluate the ability of complexity metrics to discriminate between vulnerable and non-vulnerable functions. Zimmermann et al. [31] analyzes the possibility of predicting the vulnerable components in Windows Vista by using Logistic Regression for five groups of metrics, churn, complexity, coverage, dependency, and organizational. Binary results have been evaluated with tenfold cross-validation that yields precision below 67% and recall below 21%. Meneely and William [32] study the developer-activity metrics and software vulnerabilities. Precision and recall from Bayesian network predictive model in this study are between 12%-29% and 32% to 56%, respectively. Doyle and Walden [33] study the relationships between software metrics and the vulnerable components in 14 open source web applications. Spearman’s rank correlation is computed between the metrics and security resources indicator (SRI) which is defined by the author and obtained from security scanners. This type of VPMs, which built on software metrics to pinpoint the vulnerable functions/files/components in an application, requires low inspection effort, however, they often suffer from high false positive rate.

Chowdby et al. [8] conducted a study to show complexity, coupling, and cohesion (CCC) related structural metrics are import indicators to detect vulnerable components. In this study, the authors identified 75% of the vulnerability-prone files from 52 Mozilla Firefox releases, with 30% of false positive. Moshtar et al. [35] propose net new of coupling metrics, which consider iteration of application modules, improves recall from 60.9% to 87.4% for cross-project vulnerability prediction. Shin and Williams [36], in which the authors perform binary classification using Logistic Regression with tenfold cross-validation to analyze relationship between complexity, code-churn and developer-activity (CCD) metrics and vulnerable components. 18 complexity metrics, 5 code churn metrics and fault history metric have been studied in [4]. Recall and precision from this study are 83% and 11% respectively.

Other specific applications from VPM are often based on code specific features. Perl et al. [8] analyze the effects of meta-data in the code repositories with code metrics to predict the vulnerable commits. The authors have claimed a large scale of the study with 55 C/C++ GitHub projects including 640 vulnerabilities. The precision of VCCFinder is 60% when recall is 24%. Younis et al. [37] study the relationship between software metrics and the vulnerable function in existing exploits. 183 vulnerabilities from National Vulnerability Database for Linux Kernel and Apache HTTP server have been examined in their study. Code token list has been added in the study for Stuckman et al. [38] to identify the vulnerable components in their study. The authors have claimed that the token based metrics reveal more information than the software metrics in predicting vulnerable components. Walden et al. [39] compare the predictive powers between software metrics and text mining in predicting vulnerable components. Davari et al. [40] propose an automatic vulnerability classification framework based on the features that extract from textual reports and code fixes of vulnerabilities. Li et al. [41] design a deep learning-based vulnerability detection system, Vulnerability Deep Pecker(VulDeePecker) to automatic exact features from vulnerable code fragments from one product and predict vulnerabilities in other products.

Mathematic VDM focuses on modeling discovery process of software vulnerabilities by evaluating the number of vulnerabilities with time. The existing models are Linear [9], Exponential [10], Alhazmi Malaiya Logistic (AML) [42], and effort based model. Woo et al. [43] studied both time and effort-based vulnerability discovery models based on Apache and IIS. Massacci et al. [44] conducted empirical study related to VMDs, and discovered that simplest linear model is the best appropriate choice in terms of quality and predictability for the first 6-12 months since a release data. Not only to predict the accumulated number of vulnerabilities, Johnson et al. [45] propose time between vulnerability disclosure (TBVD) to evaluate the likelihood of finding a zero-day vulnerability within a given time-frame. However, those models are specific to one application and normally large history data are needed to obtain better-fitted model.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the possible relationships between the software features and the number of vulnerabilities. To the best of our knowledge, this was so far the most comprehensive study, which contains with 780 software applications, including 6498 vulnerabilities. Seven feature selection methods are then applied to the dataset to obtain the best feature sets for the learning models. Finally, nine feature sets (seven from feature selection, one with full features and one from feature combining algorithm (PCA)) are fed into six learning models in order to predict the number of CVEs for GitHub dataset. The predictive power has been evaluated though four performance measures. This study demonstrates how vulnerabilities may be predicted based on combinations of features using machine learning techniques. The limitations of our work and future directions 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, the semantic code related features, such as CodeGadget (a number of program statements, which are semantically related to each other [41]), have not been taken into consideration in this work. A future direction is to add more features to extract from the code to the feature set and repeat the experiments to observe the evolution of the predictive power.

REFERENCES


