Keyword Extraction From Specification Documents for Planning Security Mechanisms

Jeffy Jahfar Poozhithara
Computer & Software Systems
University of Washington
Bothell, USA
jeffyj@uw.edu

Hazeline U. Asuncion
Computer & Software Systems
University of Washington
Bothell, USA
hazeline@uw.edu

Brent Lagesse
Computer & Software Systems
University of Washington
Bothell, USA
lagesse@uw.edu

***Abstract*—Software development companies heavily invest both time and money to provide post-production support to fix security vulnerabilities in their products. Current techniques identify vulnerabilities from source code using static and dynamic analyses. However, this does not help integrate security mechanisms early in the architectural design phase. We develop VDocScan, a technique for predicting vulnerabilities based on specification documents, even before the development stage. We evaluate VDocScan using an extensive dataset of CVE vulnerability reports mapped to over 3600 product documentations. An evaluation of 8 CWE vulnerability pillars shows that even interpretable whitebox classifiers predict vulnerabilities with up to 61.1% precision and 78% recall. Further, using strategies to improve the relevance of extracted keywords, addressing class imbalance, segregating products into categories such as Operating Systems, Web applications, and Hardware, and using blackbox ensemble models such as the random forest classifier improves the performance to 96% precision and 91.1% recall. The high precision and recall shows that VDocScan can anticipate vulnerabilities detected in a product's lifetime ahead of time during the Design phase to incorporate necessary security mechanisms. The performance is consistently high for vulnerabilities with the mode of introduction: *architecture and design*.**

***Index Terms*—Security, Vulnerability Prediction, CVE, CWE, Keyword Extraction, Documentation**

I. Introduction

Software projects typically undergo continuous security testing after deployment in pre-production environments which costs software development companies significant time and money [1]. Moreover, vulnerabilities detected at this stage can lead to multiple code revisions, causing unexpected delays. Proactive software maintenance engineering [2] offers a solution to eliminate security flaws *prior* to release, thereby reducing costs and potential damaging litigation to the software company [3]. A reliable mechanism to list potential vulnerabilities that are likely to be encountered based on the functional and technical specifications helps incorporate appropriate architectural design decisions such as security design patterns before the development stage.

CWE (Common Weakness Enumeration) [4] is a community-developed list of software and hardware weakness types, used to establish baselines for weakness identification, mitigation, and prevention. The CWE database contains the vulnerability description, observed examples and consequences, modes of introduction, applicable platforms and mitigation measures. This is one of the most comprehensive collections of security weaknesses currently available and is useful for identifying weaknesses across all phases of software development. While our tool focuses on the design phase, other tools such as GitHub's CodeQL[1] use it during the implementation phase. Among the vulnerabilities in CWE, early identification can be especially useful in addressing vulnerabilities where the mode of introduction is 'architecture and design' (e.g., CWE-284: Improper Access Control, CWE-693: Protection Mechanism Failure) or 'implementation' (e.g., CWE-703: Improper Check or Handling of Exceptional Conditions, CWE-707: Improper Neutralization). Such vulnerabilities may be mitigated during the design phase through the use of security mechanisms such as secure design patterns [5], [6], protection mechanisms [7], and architectural security tactics [8]. For example, CWE-284 can be addressed using Authentication Enforcer pattern, CWE-703 through Exception Shielding pattern or Safe Data Buffer, and CWE-707 through Layered Encryption or Morphed Representation patterns [9]. While the CWE database is useful for design phase detection of software security flaws, our approach is not strictly tied to it, and could be extended to utilize additional datasets that may be created in the future.

Many vulnerability prediction tools require the source code [10]–[13] to determine correlations between vulnerabilities and component characteristic(s) like code churn, complexity metrics [14], dependencies, code coverage [15], developer activity metrics [16], import statements [17], code gadgets (number of semantically related lines of code) [18], or analyzing raw source code as text [3], [19]. However as we mention above, incorporating these fixes post-development requires more time if system-wide architectural changes are required.

Machine learning-based techniques that rely on source code also suffers from overfitting when applied to cross-project vulnerability prediction, where a model is trained on one project and tested on another project (or another codebase) [11]. It relies on the unverifiable assumption that certain underlying attributes of source code will universally indicate the presence

[1] https://codeql.github.com/

of vulnerabilities, regardless of which codebase or project contains these attributes. Features based on natural language such as documentation are more likely to be universal across projects and codebases.

Vulnerability prediction tools that rely primarily on documentation are few [20]–[23]. Moreover they do not strictly enable vulnerability prediction at the design phase since they use post-development insights, source code metrics or graphs/flow diagrams generated from source code [21], [23], or are restricted to a specific programming language [11]–[13], domain or vulnerability type [20], [24], [25].

Our method, VDocScan, is novel as its only pre-requisite is specification documents that are prepared *before* product implementation begins, and it is domain and programming language agnostic. We exploit the fact that software companies create the technical and functional specification document for each product prior to development. We use these documents to find correlations between keywords (or n-grams) in these files and the vulnerabilities that were reported in CWE and CVE (Common Vulnerabilities and Exposures) [26].

Figure 1 summarizes the practical application of VDocScan. Requirements engineering refers to the first phase of the software engineering process, wherein user requirements are collected, understood, and specified for developing quality software products [27]. Once the technical and functional specifications are documented, it may be used as input to VDocScan. VDocScan evaluates the documentation against numerous vulnerability classes and outputs a list of vulnerabilities that are likely to be identified over the course of the product's lifecycle. This list may in turn be used for proactive software development and software maintenance, such as planning security mechanisms.

While there is a large body of work in defect prediction [28] and bug localization [29], [30], the body of work on vulnerability prediction is smaller [31]. The lack of a standard dataset for vulnerability prediction is a major hurdle. The rarity of vulnerabilities further complicates the situation as it leads to severe class imbalance between vulnerable and neutral software components, increasing the difficulty of building effective prediction models [11]. Yet, with even the aforementioned challenges, VDocScan predicts multiple vulnerabilities, including in closed source projects, as well as over different product types like Web applications, Hardware/Firmeware and Operating systems.

Our contributions are as follows:

1) an end-to-end vulnerability prediction technique that includes data preparation to model evaluation,
2) strategies for addressing class imbalance in the dataset,
3) an extensive dataset, mapping product specification with vulnerability reports.

Performance of machine learning-based vulnerability prediction methods is heavily influenced by the dataset used [13]. Since there is currently no dataset that maps specification with vulnerabilities, we created a dataset leveraging vulnerability reports on CVE [26] and publicly available product documentations. We created tools to automatically (1) retrieve historical vulnerability reports published on CVE, and (2) download publicly available specification documents for products listed in CVE from their respective vendor websites. The dataset created consists of 296,931 vulnerability reports from CVE comprising 52,110 products from 23,971 vendors. Specification documents for over 3,602 these products from 20 different vendors were used to evaluate the effectiveness of keyword extraction in predicting vulnerabilities.

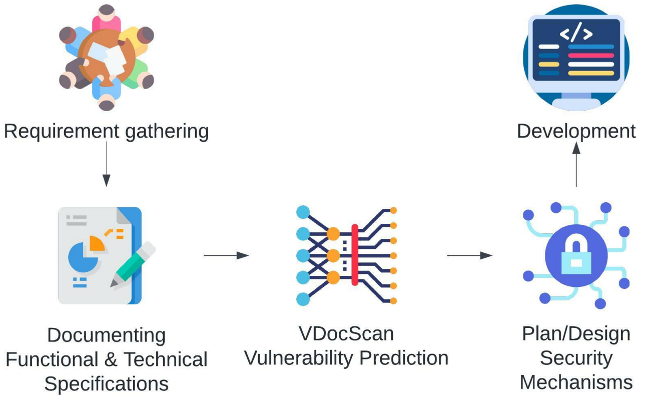

Requirement gathering
Development
Documenting Functional & Technical Specifications
VDocScan Vulnerability Prediction
Plan/Design Security Mechanisms

Fig. 1: VDocScan usage in practice

We evaluate VDocScan via the following research questions:
RQ1: Can interpretable whitebox classifiers such as decision trees make robust predictions of security vulnerabilities based on the extracted keywords?
RQ2: Can blackbox classifiers such as ensemble models improve the prediction accuracy?
RQ3: Does segregating products into categories such as Applications, Operating Systems, Hardware (embedded firmware) improve the relevance of the extracted keywords and predict better?
RQ4: Can a correlation analysis provide insights into potentially co-occurring vulnerabilities, thereby allowing simultaneous fixes?

Broadly, our results are as follows: The model was able to achieve up to 61.1% precision and 78% recall with whitebox classifiers, and up to 81.1% precision and 83.9% recall with blackbox classifiers using data from just 3,602 products. Due to the unbalanced nature of the dataset, a balanced class weighted random forest classifier was found to be the most effective binary classifier for predicting vulnerabilities. Further, we were also able to identify highly correlated vulnerabilities by analyzing over 296,931 vulnerability reports. We elaborate on these results in Section III.

II. Method

The major steps in VDocScan are Data Collection and Preparation, Model Selection and Feature Extraction, and Model Prediction/Evaluation. An overview of this approach is shown in Figure 2, where the steps are identified with dashed rectangles. The upcoming sections elaborate on each of these.

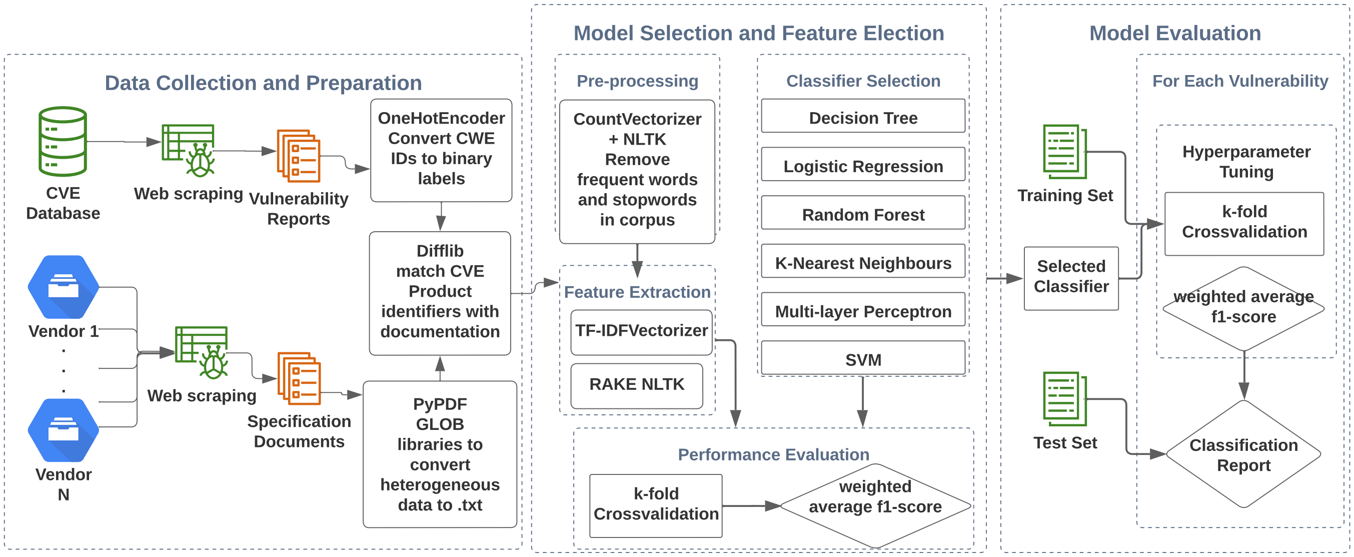


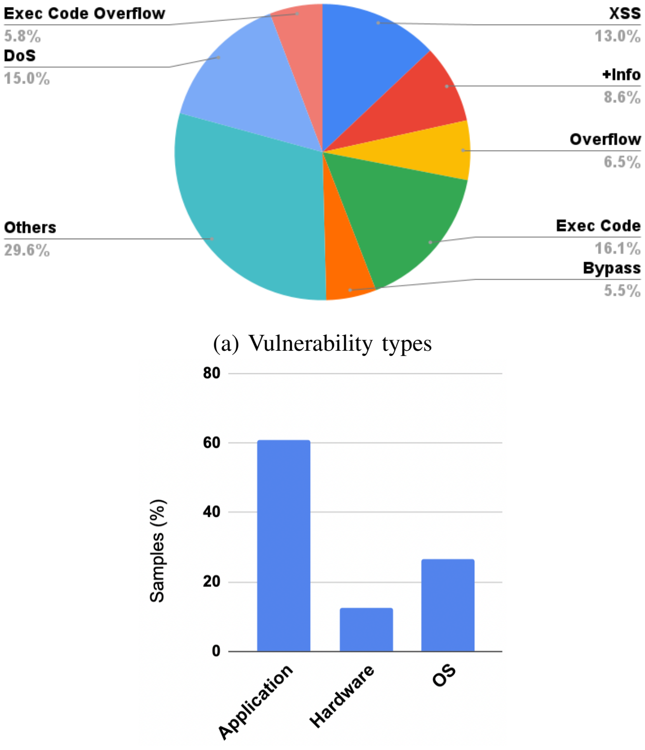
Fig. 2: Approach overview

A. Data Collection & Preparation

*1) **Data Collection**:* We created two web scraping tools to extract data from CVE and Vendor websites.
Webscraper 1: To extract vulnerability reports, we created a tool that downloads each product's metadata per vendor in the CVE website. The output is a CSV file of 52,110 products from 23,971 vendors. The CSV file was then used to further extract vulnerability reports of each of those products, resulting in a CSV file containing 296,931 vulnerability reports. The composition of vulnerability types coverage in this dataset is summarized in Figure 3.

Webscraper 2: To download the specification documents for products in the CVE data, we scrape/crawl each company's website. This required a custom website parsing logic for each vendor based on their website structure, depending on where documentations are staged. To minimize the manual effort while maximizing the number of specification documents downloaded, we identified vendors with the most number of products from the CVE Dataset produced by Webscraper 1. We further identified vendors whose websites had a central webpage linking to their products' documentation/manuals. We then created a web scraping program to apply the required security handshake protocols to fetch data (where documentation files are in HTML or JSON format) from API/Web service endpoints or to download files in .txt or .pdf formats.

Although the data parsed from CVE contains vulnerability reports from 52,110 products, due to the customization required for each vendor and the manual work to pre-process and clean the heterogeneous data collected, we only collected 7,000 unique products' documentations for evaluation. This includes documentation for products like Microsoft's Xamarin [32] and GNU tar [33]. To allow the dataset to be extended via crowd-sourcing, we make both the source code for the two webscraping tools and the current dataset available at [34] and [35] respectively.



(a) Vulnerability types

(b) Product categories

Fig. 3: Summary chart of the CVE dataset composition

Product Name Matching: Since the product name mentioned on CVE does not always match the documentation file name, we cannot directly perform text matching to map the documentations to the CVE data. While some differences such

	#Vulnerability Reports	#vendors	#products	#CVE IDs	#CWE IDs	#Vulnerability types
Initial	296931	23971	52110	111561	277	192
Filter 1: Remove missing CWE IDs	223363	18527	41028	81576	276	181
Transform1: Group by metadata	89633	18527	41028	-	276	-
Transform 2: One-hot encoding	-	18527	41028	-	276	-
Filter 2: Matching documentation found	-	18	3602	-	164	-

TABLE I: Evolution of dataset sizes through our data cleaning via the indicated filters and transformations.

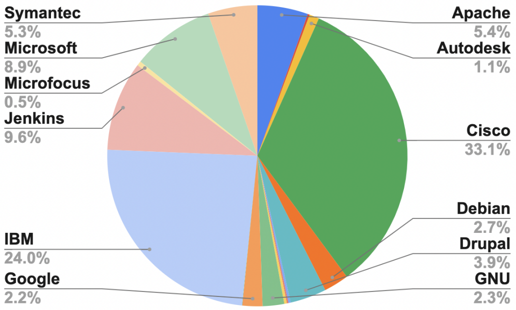


Fig. 4: Chart summarizing the composition of vendors in the documentations dataset

as capitalization mismatch may be ignored, others, such as the presence of special characters or abbreviations needs to be specially handled. To overcome this challenge, we used a text similarity scoring library: difflib [36] to identify the closest matching filenames for each product referred in the CVE data. We set a cut-off threshold of 0.6 so that a documentation file is fetched only if there is a potential match. Sample product names as per CVE and their corresponding matching filenames found in the website parsing output is shown below:

```

----Libswresample-----
['libswresample', 'libswscale', 'Audio resampler']
----Git Changelog-----
['git-changelog_', 'changelog-history_', 'gitlab-logo_']
----Gearman-----
['gearman-plugin_', 'variant_', 'vagrant_']
----Gerrit Trigger-----
['gerrit-trigger_', 'urltrigger_', 'ivytrigger_']
----Mod Pagespeed-----
['PageSpeed Module', 'Cloud Storage', 'Cost Management']
```

Although we download more than 7,000 files, due to mismatch in the filenames or due to re-branding or changing of ownership due to company buyouts, the final dataset used for evaluations contained only 3,602 products. The composition of products per vendor in the dataset is summarized in Figure 4. When the cutoff applied was too low (0.3 - 0.5), the number of false positives was high. When the cutoff applied was too high (>0.7), the number of false negatives was high. At a cutoff of 0.6, we were able to achieve a significant number of true positives with very low false positives. By ensuring that we only compare the documentation files of a vendor with the subset of the CVE dataset corresponding to the same vendor, the 214 false positives were reduced to 0. Resolving false negatives requires manual verification and mapping, however for our preliminary analysis we simply eliminate them from the dataset, see Table I.

*2) **Data Pre-Processing**:* The dataset pre-processing involves:

1) **Homogenization of file formats to plain text** - plain text or strings are required formats for classification algorithms
2) **Verification of the encoding and content of text data extracted from PDF and HTML file types** - due to differences in character encoding across filetypes, we manually verify the parsed text to ensure removal of wrongly decoded content.
3) **Transformation of the vulnerability reports to a (feature set, label) pair format** - format required by our classifier.

Gold Standard: The CVE Program identifies, defines, and catalogs publicly disclosed cybersecurity vulnerabilities [37]. Vulnerabilities are first discovered, then reported to the CVE Program. Once the reported vulnerability is confirmed by the identification of the minimum required data elements for a CVE Record, the record is published to the CVE List. CVE Records are published by CVE Program partners from around the world.

This thorough process ensures that CVE Records can be used as our gold standard dataset. While a product may have more than one vulnerability, we focus on predicting one vulnerability at a time, reducing the vulnerability prediction problem into a series of binary classification problems. This also overcomes the missing class problem that non-exhaustive datasets may introduce when directly approached as multi-class classification [38]. We further ensure the integrity of our dataset by verifying that the accuracy remains intact on multi-fold cross validation [39].

To formulate the problem as a binary classification task, we associate vulnerability reports with a 0 or 1 label. That is, instead of a dataset where each row represents a vulnerability report, we need rows to represent a specific product. We transform the dataset of vulnerability reports to a dataset of 3,602 rows where each row represents a product. We chose CWE IDs as the binary label as these IDs are not changed every year (in contrast to CVE-IDs). There were 81,576 CVE IDs for 41,028 products, whereas there were only 276 unique CWE IDs for the same data (see Table I). Among the 3,602 products that remained after the product-documentation matching, there were only 164 unique CWE IDs in the dataset. Finally, we use *one-hot encoding* [40] to convert each CWE ID to a column that receives a label of 1 if the product has the vulnerability reported. Otherwise, the column receives a zero-label. The dataset after the above transformation is shown in

	Vendor	Product	Documentation	435	682	703	691	284	693	664	707
1	GNU	Gzip	gzip reduces the size of the named files using Lempel–Ziv coding (LZ77). Whenever possible, each file is replaced by one with the extension '.gz', while keeping the same ownership ...	0	0	0	0	0	0	0	1
2	Ffmpeg	Libavcodec	The libavcodec library provides a generic encoding/decoding framework and contains multiple decoders and encoders for audio, video and subtitle streams, and several bitstream ...	0	0	0	1	0	0	1	1
3	Ffmpeg	Libswresample	The libswresample library performs highly optimized audio resampling, rematrixing and sample format conversion operations. Specifically, this library performs the following ...	0	0	1	0	0	0	0	0
4	Debian	APT	Advanced package tool, or APT, is a free-software user interface that works with core libraries to handle the installation and removal of software on Debian, and Debian-based Linux...	0	0	0	0	0	0	1	1
5	Microsoft	Xamarin	Xamarin is an open-source platform for building modern and performant applications for iOS, Android, and Windows with .NET. Xamarin is an abstraction layer that manages...	0	0	0	0	1	0	1	0

TABLE II: Sample rows from the dataset after one-hot encoding CWE IDs for binary classification

Table II.

Count Vectorizer for custom stopwords: Stopwords are words frequently encountered in text data. For text data in the English language, this list consists of words like *a, the, at, of* etc. In addition to the default stopwords list for English language, this specific dataset also contains frequent occurrences of certain technology-related terms like *technology, service, and solution* which may receive undue significance. We identify such words by using CountVectorizer to select words that have high frequency across the documentation files in the dataset. We add these words to the default stopwords list to exclude them from the extracted keywords.

B. Model Selection and Feature Extraction

CWE provides weakness information for over 900 different software and hardware quality and security issues. Out of these, only 164 CWE IDs had at least one sample in our dataset. The dataset is sparse with the most reported vulnerability (CWE-79) having only 633 samples. Evaluating the frequency distribution of vulnerability reports, the median was 9 samples per CWE ID with 75th quantile = 38 samples and 95th quantile = 282 samples. The dataset needed resampling to fewer labels/categories to improve the sampling percentage per vulnerability and thereby the accuracy of the model.

CWE has a hierarchical system of five types of abstractions that embeds relationships between the weaknesses. Four well-defined hierarchical types have been reserved, from most abstract to most specific: Pillar, Class, Base, and Variant [4]. These types correlate with the nature of information contained in the CWEs as described by dimensions such as behavior, property, technology, language, and resource. A graphical visualization of the classification of vulnerabilities to parent pillars, classes and variants is available in [41]. A segment of this graph visualizing the hierarchy under CWE-284 is shown in Figure 5.

By relabeling vulnerability reports to the highest level of abstraction (pillars), we reduced the number of labels with non-zero samples to 8 vulnerability pillars (see Figure 6) with median number of samples per vulnerability = 305, 75th and 95th percentile = 610 and 1,171 samples, respectively. Labels of child vulnerabilities were merged under parent vulnerability pillars. For example, all samples under *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')* was merged under *CWE-707: Improper Neutralization*. When a child vulnerability has multiple parent pillars, samples were added to each parents' subset. The 8 vulnerability pillars are:

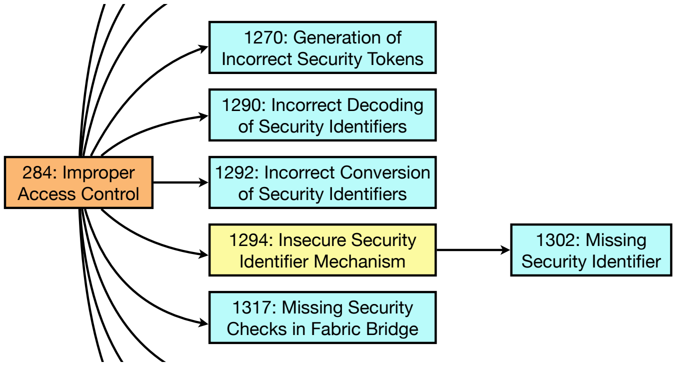

1270: Generation of Incorrect Security Tokens
1290: Incorrect Decoding of Security Identifiers
284: Improper Access Control
1292: Incorrect Conversion of Security Identifiers
1294: Insecure Security Identifier Mechanism
1302: Missing Security Identifier
1317: Missing Security Checks in Fabric Bridge

Fig. 5: Graphical Visualization of the vulnerability classification to parent pillar, classes and variants for CWE-284

1) CWE 435 - Improper Interaction Between Multiple Correctly-Behaving Entities
2) CWE 682 - Incorrect Calculation
3) CWE 703 - Improper Check or Handling of Exceptional Conditions
4) CWE 691 - Insufficient Control Flow Management
5) CWE 284 - Improper Access Control
6) CWE 693 - Protection Mechanism Failure
7) CWE 664 - Improper Control of a Resource Through its Lifetime
8) CWE 707 - Improper Neutralization

Among our parent pillars, CWE-664 contains the greatest fraction of the top 25 CWE vulnerabilities [42], which still has only 8/25, followed by CWE-707 with 5/25 vulnerabilities, and so on. Although not uniformly distributed among our 7 parent pillars, the top 25 vulnerabilities are reasonably separated in our classification.

*1) **Feature Extraction**:* The features consist of the keywords extracted from the documentation. Many tools exist

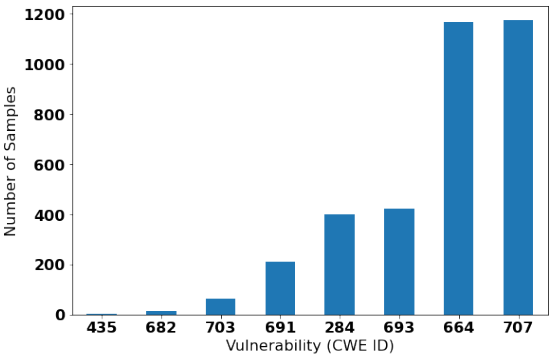


Fig. 6: Number of samples per CWE ID after grouping by parent pillars

TF-IDF Vectorizer	RAKE		
	Word Frequency	Degree to Frequency Ratio	Word Degree
fdk	ncompute	ncompute	headaches
mainframe	rfc0959	rfc0959	peap
latency	securex	securex	complexity
reviewing	rudimentary	rudimentary	x98tl
subsystems	envmon	envmon	1493
fpga	unhidden	unhidden	accompanies
fulfillment	polls	polls	dreams
pak	x98sub2	x98sub2	sequences
insights	whois	whois	006d
nexus	nfamily	nfamily	370w
js	typeset	typeset	namespaces
started	shasum	shasum	x98none
maximo	dist_lisp lisp	dist_lisp_lisp	orthis
sterling	memoir	memoir	calc
netcool	nfailures	nfailures	parking
drupal	nsuccess	nsuccess	nbd

TABLE III: Top keywords from each feature selection method

for extracting keywords. Rapid Automatic Keyword Extraction Algorithm (RAKE) [43], [44] and Term Frequency-Inverse Document Frequency (TF-IDF) vectorizer are simple methods that achieve the state of the art results in terms of both relevance and computational efficiency in many keyword extraction tasks [44].

We used the TF-IDF vectorizer and RAKE to break down the content to words/unigrams, bigrams, and trigrams. While TF-IDF vectorizer internally performs a word embedding task to convert each text sample to a feature vector, a word embedding algorithm was used to convert keywords extracted using RAKE to a feature vector. Both TF-IDF Vectorizer and RAKE support custom stopword specification [44], [45].

The top keywords extracted by TF-IDF Vectorizer and RAKE using different metrics are shown in Table III. We observed that the Degree_to_Frequency_Ratio metric of RAKE performed the best in identifying keywords that are related to different functionalities and technologies used (for example, rfc0959 is an FTP protocol). RAKE also reduced the dimensionality of the feature set over TF-IDF (23,303 vs 168,531) that led to significantly improved computation times. We chose RAKE for feature selection in our experiments.

*2) **Class Imbalance Strategies**:* The vulnerability prediction task, akin to any fraud detection or anomaly detection model, inherently faces the challenge of an imbalanced dataset. That is, the number of samples labeled as 1 (i.e., presence of the vulnerability) will be significantly smaller than the number of samples labeled as 0. This is shown in Figure 6. Out of 3,602 products, even the more common vulnerabilities such as CWE 707 and CWE 664 are found only in 1,390 and 1,361 products which account for less than 50% of the samples. When training models using imbalanced datasets, unless countermeasures are explicitly added, models tend to overestimate the likelihood of a test sample to be labeled as 0. That is, when samples are predominantly of class 0, predicting every test sample as a 0 can give a high accuracy even if other measures (precision, recall, f1-score, roc-auc) are compromised. We use the following countermeasures to address class imbalance:

Importance Sampling: To correct the prior probability calculation by a classifier, the number of samples from each class should be balanced. Down-sampling the majority class (reducing the number of samples) can help the model avoid overestimating the likelihood of the majority class [46]. This may be performed by selecting all minority class items and randomly selecting the same number of samples from the majority class. The model can then be trained on the resampled balanced dataset. To check if the model has actually learned features correctly to predict both classes, the model can be tested on Stratified Data (data with the ratio of samples of each class matching the original dataset).

Alternate Metrics: In the past few years, several new metrics have been proposed [47] which measure the classification performance on majority and minority classes independently. When executing cross-validation to benchmark the performance of different classifiers, alternate metrics such as the recall, f1-score, or Matthews correlation coefficient may be used to flag models that do not counter the class imbalance effect. In this study, we look at the weighted average f1-score in our classifier selection and hyperparameter tuning. Weighted average f1-score is calculated as the average of the f1-scores across labels, weighted by their support sizes (the number of true instances for each label), thus accounting for label imbalance.

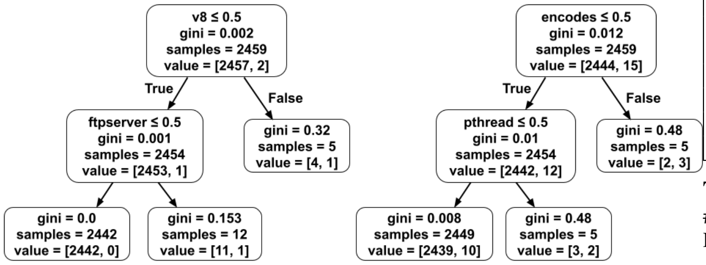
Penalized Algorithms: Some algorithms have penalties or regularization parameters to counter class imbalance [46]. For example, SVM and random forest classifiers support "balanced" or "balanced subsample" class weight mode that automatically adjusts weights inversely proportional to class frequencies in the input data.

III. Evaluation

The following experiments answer our research questions.

A. Whitebox Classifiers (RQ1)

Experiment: We evaluate the performance of the decision tree classifier in predicting the 8 vulnerability classes in the dataset. The dataset was split into training (1,967



(a) CWE 435

(b) CWE 682

Fig. 7: Decision trees for vulnerability prediction

samples) and test sets (492 samples) at a 4:1 ratio. 43,409 features (keywords extracted using RAKE NLTK) were used for model evaluation after excluding 279 frequently occuring words retrieved from the dataset using a CountVectorizer. We used hyperparameter tuning with 5-fold cross validation for each vulnerability class to find the optimal size of the tree (max_depth) and the minimum number of samples at each leaf (min_samples_leaf). Following the standard process for decision tree training, each node split was decided based on using the Gini impurity as well as the information gain [48]. Further, to account for label imbalance, we used the weighted f1-score (the weighted average of the f1-scores corresponding to the 0 and 1 labels) as the scoring metric in the hyperparameter grid search.

Results: The structure of the decision tree corresponding to each vulnerability allows us to derive meaningful insights. In a decision tree, the features close to the root are the more 'important' discriminators [49]. For CWE-435 (Improper Interaction Between Multiple Correctly-Behaving Entities), the decision tree included *ftpserver* as a feature of significant importance (see Figure 7), aligning with its vulnerability description: *An interaction error occurs when two entities have correct behavior when running independently of each other, but when they are integrated as components in a larger system or process, they introduce incorrect behaviors that may cause resultant weaknesses* [4]. Similarly, for CWE-682 (Incorrect Calculation), the decision tree included *pthread* as an important feature (Figure 7). This may be because multi-threaded programs with incorrect resource allocation are closely related to CWE-682, whose vulnerability description reads: *software performs a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management*. The decision tree rendered for CWE-284 (Improper Access Control) had *scada* as the root node. SCADA (acronym for Supervisory Control and Data Acquisition) is a computer-based system for gathering and analyzing real-time data to monitor and control equipment that deals with critical and time-sensitive materials or events [50]. Upon manual validation, it was observed that all 8 decision trees rendered had keywords relevant to the respective vulnerability with significant importance. The decision trees rendered for each vulnerability class are available as supplementary material [51]. The precision, recall, f1-score and accuracy observed in predicting the 8 vulnerability classes using the tuned hyper-parameters, are summarized in Table IV. Number of samples represent the number of 1-label entries corresponding to the vulnerability in the dataset. Note that the trivial values for precision, recall and f1-scores observed for CWE-435 and CWE-682 are due to the significantly fewer samples (just 2 and 15 samples respectively) in the dataset.

CWEID	# s	H	P	R	f1	A
435	2	gini, 2, 5	0.000	0.000	0.000	0.998
682	15	gini, 2, 5	0.000	0.000	0.000	0.992
703	62	gini, 5, 5	0.333	0.083	0.133	0.974
691	210	gini, 100, 5	0.200	0.075	0.109	0.900
284	400	gini, 20, 5	0.389	0.171	0.237	0.817
693	424	gini, 10, 5	0.350	0.171	0.230	0.809
664	1166	entropy, 20, 10	0.553	0.780	0.647	0.630
707	1174	gini, 100, 5	0.611	0.579	0.595	0.626

TABLE IV: Evaluation of the decision tree classifier
# s: Number of samples, P: Precision, R: Recall, f1: f1-score, A: Accuracy, H: Hyperparameters as (criterion, max_depth, min_samples_leaf)

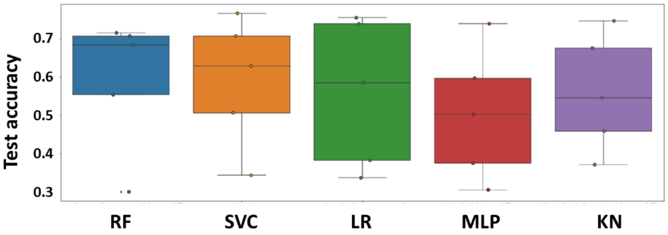


Fig. 8: Classifier selection by comparing accuracy

These results suggest that whitebox classifiers are able to predict vulnerabilities with non-trivial precision and recall when the model has over 1,000 samples to train.

B. Blackbox and Ensemble Models (RQ2)

*1) **Classifier Selection**:* We do a grid search with 5 fold cross-validation with f1-score and accuracy as our scoring metrics to identify the blackbox classifier model that works best. We compared the K-nearest neighbor classifier, logistic regression, support vector machines, multi-layer perceptron (MLP), and random forest classifier. The classifier selection involved: (1) hyperparameter tuning for each candidate classifier model, (2) grid search with 5-fold cross validation of candidate classifiers with selected hyperparameter combination.

Results: For hyperparameter tuning we used applicable class imbalance counter measures such as penalized algorithms and enabled class weighting. Further, we use the weighted f1-score as the scoring metric. The results of 5-fold cross validation with the 5 candidate classifiers (each initialized with best hyperparameter values tuned for CWE-664) using accuracy and f1-score as scoring metrics are shown in Figures 8 and 9 respectively. For each classifier, the 5 dots in the box plot shows the scoring metric value for each fold in the 5-fold cross validation. Accuracy metric was found to

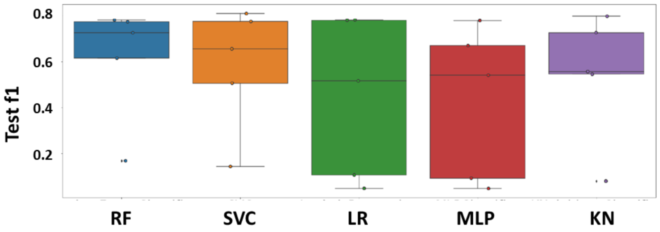
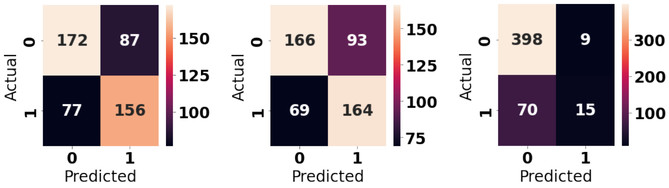


Fig. 9: Classifier selection by comparing f1-score

be heavily influenced by the dataset imbalance and we used an f1-score based evaluation to decide the best classifier. The random forest classifier trumps the other classifiers in both the mean and variance of the f1-score and we chose it for the experiments that follow. Moreover, as the random forest classifier is an ensemble of decision trees, it offers a natural extension to our whitebox model.



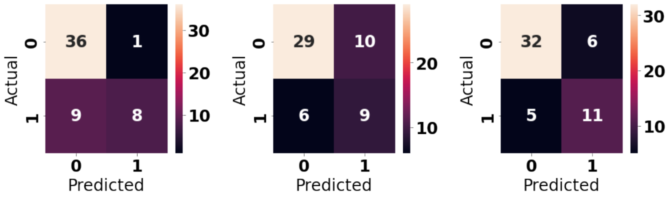
(a) CWE-664 (b) CWE-707 (c) CWE-693

Fig. 10: Confusion matrix for predicting with the random forest classifier

Vulnerability	precision	recall	f1-score	FAR	train/test size
CWE-664	0.668	0.667	0.667	0.358	1967/492
CWE-707	0.639	0.704	0.669	0.362	1967/492
CWE-693	0.811	0.839	0.8	0.375	1967/492

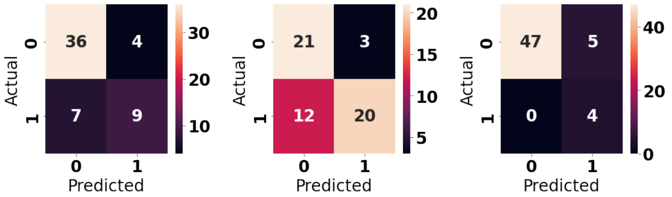
TABLE V: Performance of the random forest classifier

2) ***Best Estimator Evaluation:*** We evaluate the performance of the selected random forest classifier in predicting the top 3 vulnerability classes. We used the same test-train split ratio, feature set, and stopwords as the experiments for RQ1 to ensure a fair comparison with the whitebox classifier. For each vulnerability class, we perform a 5-fold hyperparameter grid search to tune the following parameters: 'bootstrap' (decides whether bootstrap samples or the entire dataset is used to build each tree),'max_depth', 'max_features','min_samples_leaf', and 'n_estimators' (number of trees in the forest). The confusion matrix from this evaluation is shown in Figure 10. The performance of the random forest classifier in predicting CWE-693, CWE-664, and CWE-707 is summarized in Table V. There was an average *improvement* of 52.4%, 132.8%, and 88.0% in precision, recall and f1-score respectively. The average False Alarm Rate (FAR) observed was 36.5%. Here, FAR is computed as False Positive / (False Positive + True Positive).



(a) CWE - 664 (b) CWE - 707 (c) CWE - 693

Fig. 11: Operating Systems



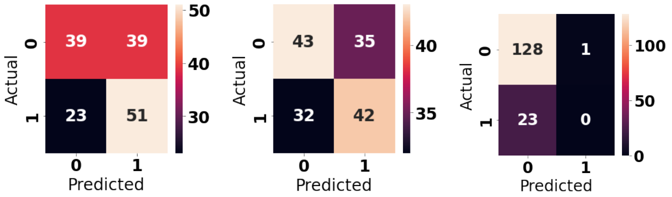
(a) CWE - 664 (b) CWE - 707 (c) CWE - 693

Fig. 12: Hardware

C. Segregating Products into Categories (RQ3)

Experiment: Instead of using all 3,602 products together in the dataset for training and testing, we split the dataset based on product types (Applications, Hardware or Operating Systems). We then extracted frequent words to be excluded for each category using Count Vectorizer. The custom stopwords list, hence generated, was used to extract keyword features using RAKE from documentations of the specific category. This is similar to a semi-supervised learning approach as category segregation allows grouping similar products together. The objective of this experiment was to evaluate if category segregation improves the relevance of the keywords extracted as they are more targeted to the domain/context.

Results: The performance of the random forest classifier in predicting CWE-664, CWE-693 and CWE-707 for each category is summarized in Figure 11,12,13, and Table VI. For Operating Systems, we observed an improvement of 11.33%, 4.75%, and 7.96% respectively for precision, recall and f1-score. Performance improvement for Hardware was 19.26%, 10.72%, and 14.89% respectively for precision, recall and f1-



(a) CWE - 664 (b) CWE - 707 (c) CWE - 693

Fig. 13: Applications

Type	Vulnerability	precision	recall	f1-score	FAR
Operating Systems	CWE-664	0.828	0.815	0.795	0.111
	CWE-707	0.73	0.704	0.713	0.526
	CWE-693	0.8	0.796	0.798	0.353
Hardware	CWE-664	0.796	0.804	0.797	0.307
	CWE-707	0.77	0.732	0.731	0.130
	CWE-693	0.96	0.911	0.926	0.555
Apps	CWE-664	0.599	0.592	0.589	0.433
	CWE-707	0.56	0.559	0.559	0.454
	CWE-693	0.719	0.842	0.776	1

TABLE VI: Precision, recall and f1-scores using a random forest model on dataset segregated by product category

score. However, for Applications, the performance deteriorated by 11.33%, 9.82% and 19.38% respectively for precision, recall and f1-score. The performance deterioration for Applications might be due to the reduced sampling percentage. While the improvement is modest, note that these results are achieved with a drastically smaller dataset when split into the above product categories. For vulnerabilities where enough samples are maintained in the dataset after splitting, there was a significant performance improvement (e.g., for CWE-693 in the Hardware category (Figure 12), the performance increased up to 96% precision, 91.1% recall and 92.6% f1-score). The improvement may be tied to the increased relevance of the extracted keywords. For example, when manually inspecting the extracted stopwords for the Hardware category, it was observed that words like 'network' (frequency: 5,513), 'power' (frequency: 5,185), 'mib' (frequency: 4,343), 'ethernet' (frequency: 3,862), 'series' (frequency: 3,828), 'ports' (frequency: 3,733) had a very high frequency, and were consequently removed from the feature list. This led to novel, less frequent, and arguably more discriminative keywords such as 'pwrinj6' (a power injector model name) becoming part of the feature set. The FAR significantly improved when segregating products into categories, for example, the FAR for CWE-664 in operating systems was only 11% and for CWE-707 in hardware was 13%, with an average FAR of 33% for both categories. FAR, however, regressed for the applications category (similar behavior to precision and recall) likely due to the reduced sampling for that category.

D. *Correlation Analysis (RQ4)*

We perform a correlation analysis to compute the pairwise correlation of columns (CWE IDs) to analyze the co-occurrence of different vulnerabilities. As correlation analysis is not constrained by documentation availability, we are able to consider all 223,363 vulnerability reports with CWE IDs (see Table I). This analysis is especially useful if there are resource or time constraints, where prediction of a subset of vulnerabilities may be desirable. If two CWE IDs are highly correlated, separate binary classification of the two labels may not be required. Moreover, the feature set (keywords extracted from the documents) can be merged to create a more fine-tuned classifier.

We were able to identify more than 30 highly correlated vulnerability pairs (correlation>0.65, p-value<0.001) where the co-occurrence was large enough for combined prediction, or where one vulnerability can be used as proxy/indication of another. We manually verified that these vulnerability pairs were not directly related (parent-child relationships) through vulnerability classification relationships. The 10 most correlated vulnerabilities are listed in Table VII. When two vulnerabilities are highly correlated, detection of one can help plan ahead to implement security mechanisms to address correlated vulnerabilities. While the high correlation of vulnerability reports may be due to insecure coding practices causing the system to be prone to multiple types of vulnerabilities, developers without domain expertise may be able to use these relationships to proactively fix a correlated vulnerability before its detection. For example, CWE-317 is a Sensitive Data Protection Vulnerability usually addressed using Information Obscurity or Secure Communication pattern [9]. If this vulnerability is detected in a system, engineers can also check for potential Logging And Audit Vulnerabilities and incorporate Secure Logger pattern where applicable.

E. *Comparison with Related Tools*

Since this is the first study that makes domain-agnostic and programming language-agnostic predictions of security vulnerabilities from specification document text, no direct comparison is possible for the precision, recall, accuracy, and f1-score measures discussed in the evaluations. Most other studies have considered predicting vulnerabilities in at most 3-5 products. For example, [18] predicted CWE 190 in Xen 4.6.0, and CWE 119 in Seamonkey 2.31 and Libav 10.2. To compare how our model performs on these products, the documentation of these 3 products were downloaded, text extracted, tokenized, and evaluated by training our model for both CWE 190 and CWE 119. We were careful to ensure that the above products were not part of our training set. Our model was able to correctly predict the presence of CWE 119 in Seamonkey and Libav. VDocScan did not detect CWE 190 in the three products. This may be due to the dataset having very low sampling size for CWE 190 (only 8 products in the dataset of 3,602 reported CWE 190).

IV. Discussion

We now revisit the research questions discussed earlier in the paper:

RQ1: Can interpretable whitebox classifiers such as decision trees make robust predictions of security vulnerabilities based on the extracted keywords?
Yes, with enough vulnerability report samples, decision trees can be trained to predict vulnerabilities while also providing interpretable tree plots. Tree plots can help engineers make informed decisions about addressing vulnerabilities and understand which features/technical dependencies cause the security flaw. Even with a small sample of 3,602 products, we were able to achieve upto 61% precision and 78% recall with decision tree classifier.

RQ2: Can blackbox classifiers such as ensemble models improve the prediction accuracy?

Vulnerability 1	Vulnerability 2	Correlation
317 - Cleartext Storage of Sensitive Information in GUI	778 - Insufficient Logging	1
98 - Improper Control of Filename for Include/Require Statement in PHP	644 - Improper Neutralization of HTTP Headers for Scripting Syntax	1
87 - Improper Neutralization of Alternate XSS Syntax	98 - Improper Control of Filename for Include/Require Statement in PHP	1
87 - Improper Neutralization of Alternate XSS Syntax	644 - Improper Neutralization of HTTP Headers for Scripting Syntax	1
313 - Cleartext Storage in a File or on Disk	317 - Cleartext Storage of Sensitive Information in GUI	1
313 - Cleartext Storage in a File or on Disk	778 - Insufficient Logging	1
299 - Improper Check for Certificate Revocation	1286 - Improper Validation of Syntactic Correctness of Input	1
805 - Buffer Access with Incorrect Length Value	1284 - Improper Validation of Specified Quantity in Input	0.91286
240 - Improper Handling of Inconsistent Structural Elements	1284 - Improper Validation of Specified Quantity in Input	0.91286
240 - Improper Handling of Inconsistent Structural Elements	805 - Buffer Access with Incorrect Length Value	0.833309

TABLE VII: Most correlated vulnerability pairs (p-value < 0.001)

The random forest classifier was able to achieve the highest average precision, recall and f1-score as well as the least variance in these metrics when evaluated on different subsamples of the dataset. With appropriate counter measures for class imbalance, random forest classifier achieved up to 81.1% precision, 83.9% recall and 80% f1-score in predicting vulnerabilities.

RQ3: Does segregating products into categories such as Applications, Operating Systems, Hardware (embedded firmware) improve the relevance of the extracted keywords and predict better?
Yes, segregating products into categories helps the feature extraction process to boost significance of keywords relevant to the domain or category. However the performance improvement upon segregation is contingent on having sufficient samples per category.

RQ4: Can a correlation analysis provide insights into potentially co-occurring vulnerabilities and thereby allow implementation of simultaneous fixes?
Yes, our current results suggest that there is significant correlation between many vulnerabilities. Highly correlated security vulnerabilities are especially helpful as detecting one vulnerability can be used as a warning for the other. Engineers can use this evaluation to plan security fixes for undetected vulnerabilities ahead of time when a correlated vulnerability is predicted in the architecture and design phase, or post-development based on source code analysis.

A. Challenges

A major challenge is the creation of the dataset as there are no existing datasets that can be re-used for this study. While we have created programs that can automatically download products to vulnerability mapping from the CVE website, downloading documentation files corresponding to products in the CVE list is difficult. Some open-source websites like GNU [52] contain text-based documentation available for most of their products that can be automatically downloaded using a webscraper. However, for most other vendors and products, the documentation cannot be downloaded in bulk as the webpage for each product differs in structure. Moreover, many companies build both software and hardware products, and identifying the products that are part of the CVE dataset is time-consuming. Most products have online HTML-based or PDF documentation. If the documentation is in PDF format, this can cause encoding issues and invalid characters due to incorrect conversion of special characters and media files embedded in the document. We addressed this issue by homogenizing the dataset into plain text format, using appropriate libraries (e.g., PyPDF for PDF files to txt conversion).

Another challenge is the computational complexity due to the large number of features. The size of specification documents can range anywhere between 10-5,000 pages. Even if we select only the top 1,000 keywords from each document, this can add up to a significant number when keywords from all documents are combined to create the final feature set. We used RAKE NLTK to extract top keywords instead of an exhaustive list of features to address the computational complexity without compromising performance.

B. Limitations

Since not all CWE labels have enough samples for the classifiers to converge with reasonable accuracy, VDocScan may not be able to make accurate predictions on all vulnerabilities. Our method is also prone to temporal gaps, as data that the model is trained on will always be older than the data it will be evaluated on. To bridge the gap, the model will need to be well calibrated using backtesting [53].

C. Threats to Validity

Internal validity: To avoid selection bias, we used all known vulnerability reports for an application as per CVE. We selected reports without regard to the severity or the type of vulnerability reported. The criteria to select applications for which documentation was downloaded was not arbitrary. We chose vendors with a large number of products, each with a significant number of vulnerability reports, to have a large dataset of product documentation and vulnerabilities for model building and prediction. We used CVE reports as the ground truth to avoid any manual bias in labeling whether the products were vulnerable.

External validity: Unlike existing studies where results might be specific to the chosen application, we cross-validated our model and evaluated it over a dataset of 3,602 products. These products are of different types (Applications, Operating Systems, Hardware/Appliances), from different domains, and written in different programming languages. Further, we consider documentation styles that range from open source projects with complete documentation and closed source proprietary projects whose documentation focuses on usage (e.g., user manual, installation guide, getting started guide etc). This diversity may allow our results to generalize across programming languages, product types, and documentation styles.

D. Future Directions

D1: A direct extension of our work is a recommendation system that suggests security mechanisms to be included in the source code based on the predicted vulnerabilities. Using design pattern detection mechanism [54] to validate whether a security design pattern has been properly implemented, VDocScan can also be used during the implementation phase.

D2: In this work we use ngram-based keyword extraction and simple models such as logistic regression and random forest models due to the limited vulnerability-specification document dataset. As ours is the first work connecting specification documents and vulnerability reports, we needed to build a new dataset and were constrained by the limited intersection between retrievable documentation and available vulnerability reports. We skipped deep learning models due to the large datasets they require to obtain comparable accuracy to simple classifiers. In time, as the dataset becomes larger, it may enable more complex and data intensive techniques such as those based on static [55] or dynamic embeddings [56] and deep learning models. This may also reduce the amount of feature engineering required, enabling a more robust end to end system.

D3: It would be of interest to study performance improvements in post-release vulnerability prediction methods from augmenting specification documents as a feature in addition to temporal information such as the version control history that they presently rely on.

V. Literature Review

While many researchers have used vulnerability databases like CVE, CWE, NVD, it is often for studying vulnerability characteristics and life cycle [13], [18], [57]–[60]. Other text classification based security vulnerability identification systems rely on vulnerability descriptions as its input [61]. We use vulnerability databases as the ground truth just as in these works, but differ in that our method takes specification documents as input. This approach is similar in spirit to [62]–[64] that apply ML techniques on specification documents. However, we are the first to apply ML techniques on specification documents for vulnerability prediction.

Other works (e.g. [20], [65]–[69]) identify vulnerabilities in source code based on code snippet samples. They primarily study the influence of various factors on vulnerability detection such as the dataset source, feature extraction, class imbalance and vectorization methods, the classifier choice, user-defined name replacement, etc. [12] proposes a hybrid technique based on combining N-gram analysis and feature selection algorithms for predicting vulnerable software components where features are defined as continuous sequences of tokens in source code files, i.e, Java class files. While these methods perform well in detecting security issues using source code, they do not enable making architectural and design decisions to avoid these security vulnerabilities before the development phase.

VI. Conclusion

While no predictive model can be expected to be 100% accurate, this tool is expected to be used to provide checkpoints for software engineers. Our objective is to *reduce* the number of vulnerabilities that are detected post release. The implication of false positives are less severe as it only leads to more secure coding practices. False negatives, however, are lost opportunities to fix the vulnerabilities ahead of time that could have saved the company both time and money. Our correlation analysis may also be used by developers to watch out for correlated vulnerabilities while making architectural/design decisions. When a vulnerability is detected post-development, our correlation analysis can also be used to proactively correct correlated vulnerabilities before they can be exploited by malicious actors.

Although many organizations have established vulnerability description databases for information on security vulnerabilities, the differences in their descriptions make it difficult to apply security precautions [70], [71]. This study also enables developers without expertise in cybersecurity to make design decisions that mitigate security vulnerabilities. Early detection of vulnerabilities using VDocScan may also benefit Open Source users from being at a disadvantage due to disclosure latency while OSS developers make silent fixes [72].

Acknowledgment

The authors thank Vaishakh Ravindrakumar (University of California San Diego), Alex Christensen, Martin Rudzis, Dinh Liu, Aidar Kurmanbek-Uulu, Matthew Hewitt, Harminder Simplay and Sayali Kudale of the University of Washington Bothell for their assistance with performing evaluations and reviewing results. This effort is partly supported by the University of Washington Bothell Computing and Software Systems (CSS) Division Project & the CSS Division Graduate Research funds.

References

[1] R. Kumar and R. Goyal, "Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC)," *Computers & Security*, vol. 97, p. 101967, 2020.

[2] N. Hanebutte and P. W. Oman, "Software vulnerability mitigation as a proper subset of software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 6, pp. 379–400, 2005.

[3] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proc of the 4th Int'l Workshop on Security Measurements and Metrics*, 2012, pp. 7–10.

[4] MITRE, "CWE - common weakness enumeration," https://cwe.mitre.org/, 2006, (accessed December 13,2020).

[5] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, "Secure design patterns," Carnegie Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 2009.

[6] X. Wang, "A secure computing platform for building automation using microkernel-based operating systems," Ph.D. dissertation, University of South Florida, 2018.

[7] A. K. Jones and R. J. Lipton, "The enforcement of security policies for computation," in *Proc of the 5th ACM Symposium on Operating Systems Principles*, 1975, pp. 197–206.

[8] J. C. Santos, A. Peruma, M. Mirakhorli, M. Galstery, J. V. Vidal, and A. Sejfia, "Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird," in *Proc of the IEEE Int'l Conf on Software Architecture (ICSA)*. IEEE, 2017, pp. 69–78.

[9] P. Anand, J. Ryoo, and R. Kazman, "Vulnerability-based security pattern categorization in search of missing patterns," in *Proc of the 9th Int'l Conf on Availability, Reliability and Security*. IEEE, 2014, pp. 476–483.

[10] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "DeepCVA: Automated commit-level vulnerability assessment with deep multi-task learning," in *Proc of the 36th IEEE/ACM Int'l Conf on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[11] 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, 2016.

[12] Y. Pang, X. Xue, and A. S. Namin, "Predicting vulnerable software components through n-gram analysis and statistical feature selection," in *Proc of the 14th IEEE Int'l Conf on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 543–548.

[13] W. Zheng, J. Gao, X. Wu, F. Liu, Y. Xun, G. Liu, and X. Chen, "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study," *Journal of Systems and Software*, vol. 168, p. 110659, 2020.

[14] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proc of the 41st IEEE/ACM Int'l Conf on Software Engineering (ICSE)*. IEEE, 2019, pp. 60–71.

[15] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proc of the 3rd Int'l Conf on Software Testing, Verification and Validation*. IEEE, 2010, pp. 421–428.

[16] 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, 2010.

[17] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc of the 14th ACM Conf on Computer and Communications Security*, 2007, pp. 529–540.

[18] 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," *arXiv preprint arXiv:1801.01681*, 2018.

[19] V.-A. Nguyen, V. Nguyen, T. Le, Q. H. Tran, D. Phung *et al.*, "ReGVD: Revisiting graph neural networks for vulnerability detection," in *Proc of the 44th IEEE/ACM Int'l Conf on Software Engineering: Companion Proc.* IEEE, 2022, pp. 178–182.

[20] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang, K. Chen, and W. Zou, "Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis," in *Proc of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 747–764.

[21] M. Ghorbanzadeh and H. R. Shahriari, "Detecting application logic vulnerabilities via finding incompatibility between application design and implementation," *IET Software*, vol. 14, no. 4, pp. 377–388, 2020.

[22] C. Batur Şahin and L. Abualigah, "A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection," *Neural Computing and Applications*, vol. 33, no. 20, pp. 14 049–14 067, 2021.

[23] S. Salva and S. R. Zafimiharisoa, "APSET, an android application security testing tool for detecting intent-based vulnerabilities," *Int'l Journal on Software Tools for Technology Transfer*, vol. 17, no. 2, pp. 201–221, 2015.

[24] C. Shou, I. B. Kadron, Q. Su, and T. Bultan, "CorbFuzz: Checking browser security policies with fuzzing," in *Proc of the 36th IEEE/ACM Int'l Conf on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 215–226.

[25] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proc of the 35th IEEE/ACM Int'l Conf on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1029–1040.

[26] MITRE, "CVE - common vulnerabilities and exposures," https://cve.mitre.org/, 1999, (accessed December 13,2020).

[27] D. Pandey, U. Suman, and A. K. Ramani, "An effective requirement engineering process model for software development and requirements management," in *Proc of the Int'l Conf on Advances in Recent Technologies in Communication and Computing*. IEEE, 2010, pp. 287–291.

[28] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, "Software defect prediction based on kernel PCA and weighted extreme learning machine," *Information and Software Technology*, vol. 106, pp. 182–200, 2019.

[29] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proc of the 34th Int'l Conf on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[30] R. Widyasari, G. A. A. Prana, S. A. Haryono, S. Wang, and D. Lo, "Real world projects, real faults: evaluating spectrum based fault localization techniques on python projects," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–50, 2022.

[31] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[32] Microsoft, "What is xamarin," https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin, (accessed August 30,2022).

[33] GNU, "GNU tar 1.34," https://www.gnu.org/software/tar/manual/tar.html, (accessed August 30,2022).

[34] J. J. Poozhithara, "Artifacts for Keyword Extraction From Specification Documents for Planning Security Mechanisms," May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7578926

[35] ——, "Artifacts for Keyword Extraction From Specification Documents for Planning Security Mechanisms," May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7578909

[36] T. Python Software Foundation, "difflib — helpers for computing deltas," https://docs.python.org/3.8/library/difflib.html, 2019, (accessed May 24,2021).

[37] MITRE, "CVE program overview - process," https://www.cve.org/About/Process, 2022, (accessed August 30,2022).

[38] M. M. Dundar, E. D. Hirleman, A. K. Bhunia, J. P. Robinson, and B. Rajwa, "Learning with a non-exhaustive training dataset: a case study: detection of bacteria cultures using optical-scattering technology," in *Proc of the 15th ACM SIGKDD Int'l Conf on Knowledge Discovery and Data Mining*, 2009, pp. 279–288.

[39] T. S. Wiens, B. C. Dale, M. S. Boyce, and G. P. Kershaw, "Three way k-fold cross-validation of resource selection functions," *Ecological Modelling*, vol. 212, no. 3-4, pp. 244–255, 2008.

[40] D. Harris and S. L. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[41] MITRE, "CWE relationship graph visualization," https://cwe.mitre.org/data/pdf/1000_abstraction_colors.pdf, 2022, (accessed August 30,2022).

[42] ——, "2022 CWE top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022, (accessed February 5,2022).

[43] S. J. Rose, V. L. Crow, N. O. Cramer *et al.*, "Rapid automatic keyword extraction for information retrieval and analysis," Pacific Northwest National Lab.(PNNL), Richland, WA (United States), Tech. Rep., 2012.

[44] S. Rose, D. Engel, N. Cramer, and W. Cowley, "Automatic keyword extraction from individual documents," *Text Mining: Applications and Theory*, vol. 1, pp. 1–20, 2010.

[45] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for IDF," *Journal of Documentation*, 2004.

[46] N. Japkowicz, "The class imbalance problem: Significance and strategies," in *Proc of the Int'l Conf on Artificial Intelligence*, vol. 56. Citeseer, 2000.

[47] C. D. Manning, P. Raghavan, and H. Schütze, "Probabilistic information retrieval," *Introduction to Information Retrieval*, pp. 220–235, 2009.

[48] L. E. Raileanu and K. Stoffel, "Theoretical comparison between the gini index and information gain criteria," *Annals of Mathematics and Artificial Intelligence*, vol. 41, no. 1, pp. 77–93, 2004.

[49] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[50] S. A. Boyer, *SCADA: supervisory control and data acquisition*. Int'l Society of Automation, 2009.

[51] J. J. Poozhithara, "Vdocscan - dataset," May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6526011

[52] GNU, "GNU operating system," https://www.gnu.org/manual/manual.html, 2020, (accessed May 24,2021).

[53] R. Giacomini and I. Komunjer, “Evaluation and combination of conditional quantile forecasts,” *Journal of Business & Economic Statistics*, vol. 23, no. 4, pp. 416–431, 2005.

[54] J. Poozhithara, H. Asuncion, and B. Lagesse, “Towards lightweight detection of design patterns in source code,” in *Proc of the 34th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2022, pp. 95–99.

[55] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” *arXiv preprint arXiv:1310.4546*, 2013.

[56] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[57] C. Elbaz, L. Rilling, and C. Morin, “Automated keyword extraction from ”one-day” vulnerabilities at disclosure,” in *Proc of the NOMS IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[58] S. Neuhaus and T. Zimmermann, “Security trend analysis with CVE topic models,” in *Proc of the 21st IEEE Int’l Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 111–120.

[59] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “Vulpecker: an automated vulnerability detection system based on code similarity analysis,” in *Proc of the 32nd Annual Conf on Computer Security Applications*, 2016, pp. 201–213.

[60] X. Li, J. Chen, Z. Lin, L. Zhang, Z. Wang, M. Zhou, and W. Xie, “A mining approach to obtain the software vulnerability characteristics,” in *Proc of the 5th Int’l Conf on Advanced Cloud and Big Data (CBD)*. IEEE, 2017, pp. 296–301.

[61] S. Yitagesu, Z. Xing, X. Zhang, Z. Feng, X. Li, and L. Han, “Unsupervised labeling and extraction of phrase-based concepts in vulnerability descriptions,” in *Proc of the 36th IEEE/ACM Int’l Conf on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 943–954.

[62] F. Iwama, T. Nakamura, and H. Takeuchi, “Constructing parser for industrial software specifications containing formal and natural language description,” in *Proc of the 34th Int’l Conf on Software Engineering (ICSE)*, 2012, pp. 1012–1021.

[63] P. R. Anish, P. Lawhatre, R. Chatterjee, V. Joshi, and S. Ghaisas, “Automated labeling and classification of business rules from software requirement specifications,” in *Proc of the 44th IEEE/ACM Int’l Conf on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 53–54.

[64] S. Imtiaz, M. R. Amin, A. Q. Do, S. Iannucci, and T. Bhowmik, “Predicting vulnerability for requirements,” in *Proc of the 22nd IEEE Int’l Conf on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2021, pp. 160–167.

[65] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, “A comparative study of deep learning-based vulnerability detection system,” *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[66] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, “Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey,” *Information Security Technical Report*, vol. 14, no. 1, pp. 16–29, 2009.

[67] C. LeDoux and A. Lakhotia, “Malware and machine learning,” in *Intelligent Methods for Cyber Warfare*. Springer, 2015, pp. 1–42.

[68] A. Souri and R. Hosseini, “A state-of-the-art survey of malware detection approaches using data mining techniques,” *Human-centric Computing and Information Sciences*, vol. 8, no. 1, pp. 1–22, 2018.

[69] Z. Xu, S. Li, J. Xu, J. Liu, X. Luo, Y. Zhang, T. Zhang, J. Keung, and Y. Tang, “LDFR: Learning deep feature representation for software defect prediction,” *Journal of Systems and Software*, vol. 158, p. 110402, 2019.

[70] K. Shi, Y. Dai, and J. Xu, “Construction of a security vulnerability identification system based on machine learning,” *Journal of Sensors*, vol. 2020, 2020.

[71] L. A. B. Sanguino and R. Uetz, “Software vulnerability analysis using CPE and CVE,” *arXiv preprint arXiv:1705.05347*, 2017.

[72] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, “Finding a needle in a haystack: Automated mining of silent vulnerability fixes,” in *Proc of the 36th IEEE/ACM Int’l Conf on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.