# Automated Identification of Libraries from Vulnerability Data

Yang Chen
Veracode
ychen@veracode.com

Andrew E. Santosa
Veracode
asantosa@veracode.com

Asankhaya Sharma
Veracode
asharma@veracode.com

David Lo
Singapore Management University
davidlo@smu.edu.sg

## ABSTRACT

*Software Composition Analysis* (*SCA*) has gained traction in recent years with a number of commercial offerings from various companies. SCA involves *vulnerability curation* process where a group of security researchers, using various data sources, populate a database of open-source library vulnerabilities, which is used by a scanner to inform the end users of vulnerable libraries used by their applications. One of the data sources used is the *National Vulnerability Database* (*NVD*). The key challenge faced by the security researchers here is in figuring out which libraries are related to each of the reported vulnerability in NVD. In this article, we report our design and implementation of a machine learning system to help identify the libraries related to each vulnerability in NVD.

The problem is that of *extreme multi-label learning* (*XML*), and we developed our system using the state-of-the-art FastXML algorithm. Our system is iteratively executed, improving the performance of the model over time. At the time of writing, it achieves $F_1$@1 score of 0.53 with average $F_1@k$ score for $k = 1, 2, 3$ of 0.51 ($F_1@k$ is the harmonic mean of *precision@k* and *recall@k*). It has been deployed in Veracode as part of a machine learning system that helps the security researchers identify the likelihood of web data items to be vulnerability-related. In addition, we present evaluation results of our feature engineering and the FastXML tree number used. Our work formulates for the first time library name identification from NVD data as XML and it is also the first attempt at solving it in a complete production system.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering** → *Software maintenance tools.*

## KEYWORDS

application security, open source software, machine learning, classifiers ensemble, self training

**ACM Reference Format:**
Yang Chen, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2019. Automated Identification of Libraries from Vulnerability Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Open-source libraries are critical to modern information infrastructure, which relies heavily on software written using open-source dependencies, such as those in Maven central, npmjs.com, and PyPI. As with any software, however, open-source libraries may contain security vulnerabilities. *Software Composition Analysis* (*SCA*) automatically identifies vulnerable versions of the dependencies used in an application, so that developers can continue using open-source libraries with peace of mind. SCA has gained traction in recent years with a number of commercial offerings from various companies [1, 9–11]. The design of a state-of-the-art SCA product is discussed in a recent article by Foo et al. [20].

Figure 1 depicts a typical SCA workflow. SCA helps developers by discovering vulnerable libraries used by their application by matching the detected dependencies of the application with a database of vulnerable libraries. SCA importantly involves a *vulnerability curation* process where a group of security researchers populate the database with data from various sources. Most relevant to our work is the *National Vulnerability Database* (*NVD*) data source. Each NVD entry includes a unique *Common Vulnerability Enumeration* (*CVE*) identification number. a vulnerability description, *Common Platform Enumeration* (*CPE*) configurations, and references (web links). Each CPE configuration is a regular expression that identifies a set of CPE names. Each name in turn identifies an information technology system, software, or package related to the vulnerability. Unfortunately, these information may not explicitly identify the vulnerable library. For example, Figure 2 shows the words that are included in the report for CVE-2015-7318, that we extracted automatically from its web page [5]. This vulnerability is included in the Zope2 Python library of Plone content management system, however, Zope2 is nowhere mentioned in the text, as well as in the original NVD record [5]. As another example, Figure 3 shows the text that we automatically extracted from the web page of CVE-2011-0448 [4]. Although the Ruby on Rails Active Record library has the vulnerability, it is neither mentioned in the text data nor in the NVD record [4]. Within the context of SCA vulnerability curation process, therefore, the key challenge faced by the SCA security researchers here is in figuring out which libraries are related to each of the reported vulnerability in the NVD from these data.

*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

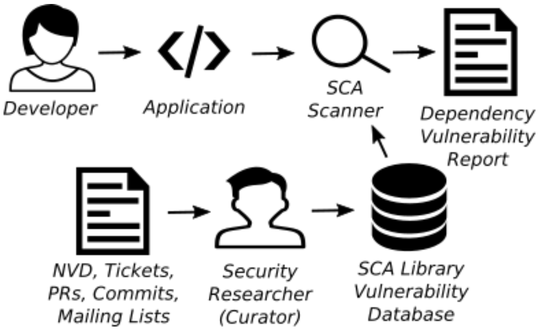
Developer
Application
SCA Scanner
Dependency Vulnerability Report
NVD, Tickets, PRs, Commits, Mailing Lists
Security Researcher (Curator)
SCA Library Vulnerability Database

**Figure 1: Software Composition Analysis**

> 2015 allows responses bugzilla.redhat.com show_bug.cgi http oss 16 header lists 09 headers injection 22 www.openwall.com plone hotfix plone.org 20150910 1264796

**Figure 2: Text from CVE-2015-7318**

> pipermail 2 source advisories lists.fedoraproject.org package announce secunia.com function 0 attacks www.vupen.com english securitytracker.com output releases sql argument 11 2011 4 april injection 3 8 group values rails groups.google.com arguments ruby rubyonrails weblog.rubyonrails.org integer limit msg dmode gplain 057650.html 0877 4e19864cf6ad40ad 43278 1025063 new

**Figure 3: Text from CVE-2011-0448**

In this article, we report an automated prediction system using machine learning to compute the related *library names* for any given CVE id. It implements the function

$$identify : NVD \longrightarrow \mathcal{P}(L) \tag{1}$$

with *NVD* the set of CVE ids in the NVD, and $\mathcal{P}(L)$ the powerset of $L$, the set of library names that identify the libraries. For training our machine learning models, we collect training data from NVD and obtained thousands of NVD vulnerability records with their library names from SCA database of library vulnerabilities, manually curated by security researchers in years. The machine learning task is in essence an *extreme multi-label text classification* (*XMTC*), which is more generally known as *extreme multi-label learning* (*XML*). For training and prediction, we use the state-of-the-art FastXML algorithm [28]. Our work formulates for the first time library name identification from NVD data as XML and it is also the first attempt at solving it.

In summary, our main contribution is the application of the FastXML [28] approach for mapping vulnerability data to library names in the context of SCA. As demonstrated by our examples of Figures 2 and 3, our problem here is in relating a query to a library whose name that may not even be mentioned in the query. Hence, the problem is different to that of information retrieval, in particular *learning-to-rank* [25]. In this article, we describe our approaches for data collection, feature engineering, and model training and validation. We also present evaluation results of our design choices in data preparation, and the FastXML tree number used. In addition, we present a case study model evaluation, where at the time of writing, our model achieves $F_1@1$ score of 0.53 with average $F_1@k$ score for $k = 1, 2, 3$ of 0.51 ($F_1@k$ is the harmonic mean of *precision@k* and *recall@k*, all of which are defined in Section 4.2).

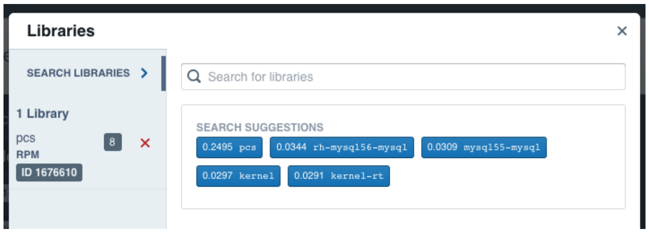

**Figure 4: Recommendation System Screenshot**

Our system has been deployed at Veracode as part of a larger system that helps the security researchers identify the likelihood of web data items to be vulnerability-related. The prediction facility is packaged as a web service, where the inputs are vulnerability description of the given CVE id, its CPE configurations, and its references, and it responds to an input with a ranked list of library names, with a *score* is attached to each library name in the list. Based on the scores, we can select the top-$k$ library names. The prediction results are made available to the researchers via a web user interface as exemplified in Figure 4.

We first provide some background in Section 2, including our vulnerability curation system for SCA, XML, and FastXML. We next describe our data gathering and feature engineering efforts in Section 3. We discuss our core approach in Section 4, and present experimental evaluations of our approach in Section 5. We then discuss a model deployment case study in Section 6, followed by a discussion on the threats to validity in Section 7. We present related work in Section 8, and conclude our article in Section 9.

## 2 BACKGROUND

### 2.1 Software Composition Analysis Vulnerability Curation System

In this article we report an enhancement to an existing machine-learning-based vulnerability curation system already used in production at Veracode [47]. Figure 5 shows the overall structure of the system of the system. The core of this system is a suite of stacking ensemble models [56] that predict whether a data item is related to vulnerability or not. The system is executed iteratively, where new models are trained monthly using more and updated input data at each iteration. The data sources fed into system at each iteration include the NVD, Jira tickets, Bugzilla reports, Github issues, PRs, and commits, as well as emails. The data from internet are first cleaned, and then after the feature extraction and selection process, existing production models are used to perform prediction on the input data to decide whether each data item is related to vulnerability or not. The result of the prediction is given to a team of security researchers to make the final decision, resulting in new entries in the library vulnerability database. The new entries are then used to train and validate new models for use in production for the next iteration.

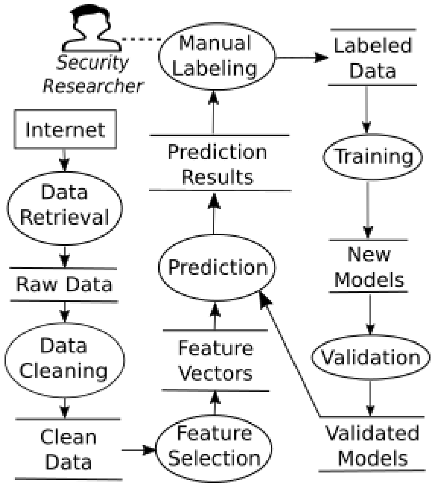

**Figure 5: Data Flow Diagram (DFD) of A Machine-Learning-Based Vulnerability Curation System. The system iteratively improves the prediction models with the labeling by the security researchers as the only manual task.**

Our work enhances the system further by automatically providing recommendation to the security researchers using a machine-learning model on the related library names for each new entry in the NVD. The security researchers review this recommendation and manually decide on the final set of library names for entry. The new association is then added into the SCA database which is then used in the subsequent iterations to train new models.

## 2.2 XML and FastXML

XML (or XMTC in natural language processing) is a problem to classify input data with multiple tags or labels [14]. Our problem can be considered as an instance of XML. XML is different from normal *multi-label learning* (*ML*) due to the extremely large number of labels for XML. XML is also different from *multi-class* classification [44], where the objective is to tag each data item with a single tag [14]. Liu et al. summarizes the main challenges in XML [24]:

1. **Severe data sparsity.** The majority of the labels have very few training instances associated with them, making it difficult to learn the dependency patterns among labels reliably.
2. **Computational costs.** The costs of both training and testing of independent multi-class classifiers (i.e., the *binary relevance* (*BR*) method [26]) is prohibitive due to the large number of labels, possibly reaching millions.

There are several approaches to XML in the literature. The *embedding* methods perform *compression* to reduce the dimensionality of the label vector space. The approaches in this category include WSABIE [41], SLEEC [14], and AnnexML [34]. SLEEC is reported to slightly outperform FastXML [28] (the algorithm that we use) on LSHTC4, a large-scale benchmark for text classification using Wikipedia dataset [27]. In embedding-based approach, however, the models can be costly to train [41] and the prediction can also be slow even for small embedding dimensions [14], as it requires to decompress a prediction to the labels in the original vector space. More importantly, because of data sparsity [24, 34], the critical assumption that the training label matrix is low-rank, is violated in almost all real world applications [14]. This entails a heavy loss of information in the compression.

*Deep learning* approaches to XMTC include that of Liu et al. [24], You et al. [50], and Zhang et al. [53]. Shah et al. also report the application of *fastText* [23], which is a deep learning approach, to the product matching problem, demonstrating that the approach is efficient to train [31]. Deep learning has been shown to have competitive results compared to FastXML (the algorithm that we use) [24, 53], but deep learning is not applicable to us since we only have about more than 7,000 labeled data items (see Tables 1 and 7), which is insufficient for deep learning.

In the *tree-based* methods, the input or label space is arranged in a tree hierarchy, where the root usually represents the whole dataset. Agrawal et al. [12] show that using *multi-label random forest* (*MLRF*) classifier results in significantly better bid phrase recommendations than ranking-based techniques. *Label partitioning for sublinear ranking* (*LPSR*) approach partitions the input space and assigns labels to the partitions taking into account the ranking provided by an *original* scorer of the label assignment. This achieves validation time complexity sublinear in the size of the label space [42]. Tree-based methods generally perform better than embedding-based methods, and we thus base our work on the state-of-the-art tree-based algorithm FastXML, which is shown to have much less training costs than MLRF or LPSR [28], and is often considered as state of the art for comparison with other approaches [24, 53]. The main difference of FastXML to MLRF and LPSR is in the tree node partitioning: FastXML's node partitioning formulation directly optimizes a rank sensitive loss function called *normalized DCG* (*nDCG*) over all the labels. *DCG* (*discounted cumulative gain*) is a measure to quantify the quality of ranking. Given a rank $p$ of an item, its *DCG* is $DCG(p)$ [40]. Normalized DCG (nDCG) is $DCG(p)$ normalized to $[0, 1]$ using an *ideal* DCG (*IDCG*), which is the maximum of the $DCG(p)$. That is, $nDCG(p) = DCG(p)/IDCG(p)$. The use of nDCG in FastXML leads to more accurate predictions over MLRF's Gini index or LPSR's clustering error. FastXML uses an efficient alternating minimization algorithm to optimize $nDCG(p)$ that converges in a finite number of iterations. Jain et al. proposes PfastXML that improves FastXML by replacing optimization of *nDCG* with minimization of propensity-scored loss *precision@k* or *nDCG@k* [22]. They also propose PfastreXML algorithm which further re-ranks the predictions of PfastXML using classifiers that take into account *tail* labels which occur infrequently. The implementation of FastXML that we use implements FastXML, as well as PfastXML and PfastreXML [2].

There is also a newer category of XML algorithms called the *sparse model* methods, such as *PD-Sparse* [49], with parallelized versions by Yen et al. [48] and Babbar and Schölkopf [13], however, even when considering these methods, FastXML's training and prediction times are still competitive for our purpose.

# 3 INPUT DATA

## 3.1 Data Sources

The data that we use for model training come from two sources: the NVD and the SCA library vulnerability database. We obtain vulnerability data from NVD and library names from the vulnerability database for constructing the training dataset, which maps NVD data to the library names (Equation 1). We use this dataset to train our models. The training input is a finite set of labeled data $d_{train}$, which is a set of pairs of vectors $\{(\vec{x}_i, \vec{y}_i)_{i=1}^N\}$ with $N$ the number of all pairs, and where $\vec{x}_i \in \vec{X}$ is the feature vector of an input data item, and $\vec{y}_i \in \vec{Y}$ is the label vector of length $|L|$ (this equals 4,682 for our experiments dataset of Section 5) when $L$ is the set of labels, and where $\vec{y}_i$, for a given subset $D \subseteq L$ (where $|D|$ is typically orders of magnitude smaller than $|L|$), is defined by an indicator function $1_D^i : L \longrightarrow \{0, 1\}$ such that $\vec{y}_{ij} = 1_D^i(L_j)$. Therefore, $\vec{y}_{ij} = 1$ if and only if the input $\vec{x}_i$ is labeled with the library name $L_j$, for possibly multiple values of $j$ $(1 \leq j \leq |L|)^1$. From the NVD data we obtain the features for $\vec{x}_i$s and the CVE ids, while from the SCA vulnerability database we obtain a mapping of the CVE ids to the library names, which we then use to obtain the $\vec{y}_i$s by matching the CVE ids. As an example, for CVE-2015-7318, the $\vec{x}_i$ part is the vectorization of the input text in Figure 2 while the $\vec{y}_i$ encodes the set $D = \{\texttt{zope2}, \texttt{plone}\}$ of library names, with $\vec{y}_{ij} = 1$ if and only if $j$ is the index of `zope2` or `plone` in $L$.

We implement a script to automatically download JSON data files from NVD [6] from 2002 to 2019. Each entry in the JSON file corresponds to an NVD web page [7]. We note that the content of each file may be different when downloaded at different times due to updates to the NVD. The JSON files include all the features (for $\vec{x}_i$s) that we use to train our models. In particular, we use the CVE ids to construct the function $f_{NVD} : NVD \longrightarrow \vec{X}$, where $NVD$, as in Equation 1, is the set of CVE ids, and $\vec{X}$ is the set of all input feature vectors. Each $\vec{y}_i$ maps $\vec{x}_i$ into a set of libraries identified by library names. We obtain this mapping from SCA vulnerability database, which has been manually curated by security researchers in years. Each library name is a pair of *coordinates* of the library recorded in the vulnerability database. The SCA vulnerability database provides an API which, given a CVE id, returns the coordinates of all libraries already identified to be related. The coordinates of a library consist of a *first coordinate* (*coordinate1*), and an optional *second coordinate* (*coordinate2*). For example, a Maven Java library such as Jackson Databind has its *group identifier* `com.fasterxml.jackson.core` and its *artifact identifier* `jackson-databind` as respectively the first and second coordinates. In this case, we conjoin both coordinates into the form "*coordinate1 coordinate2*" as the name for the library, which for our Jackson Databind case is "`com.fasterxml.jackson.core jackson-databind`." Some libraries, however, only have the first coordinate, which is directly used as its name. One CVE id may be mapped to a multiple of library names. When $L$ is the set of all library names, from the SCA vulnerability database we get the function $f_{SCA} : NVD \longrightarrow \mathcal{P}(L)$, which is a function that maps the CVE ids in the NVD to the subsets of library names. We note that the support of $f_{SCA}$ is a subset of $NVD$. This is because Veracode SCA has a limited focus on open-source libraries and a number of programming languages.

We finally combine $f_{NVD}$ and $f_{SCA}$ together by matching CVE ids to build our training data. From both $f_{NVD}$ and $f_{SCA}$ we construct a function $f_{NVD\text{-}SCA} : \vec{X} \longrightarrow \mathcal{P}(L)$ that directly maps the input feature vectors to the subset of labels, defined as the following set:

$$\{(\vec{x}, D) \mid \exists id \in NVD \cdot \vec{x} = f_{NVD}(id) \wedge D = f_{SCA}(id)\}$$

Finally, our training dataset is thus the finite function $d_{train} : \vec{X} \longrightarrow \vec{Y}$ defined as the set

$$\{(\vec{x}, \vec{y}) \mid D = f_{NVD\text{-}SCA}(\vec{x}) \wedge (\forall 1 \leq j \leq |L| \cdot \vec{y}j = 1_D(L_j))\}. \qquad (2)$$

This dataset is passed on to FastXML [28] to train new models.

```
text = re.sub(r"[^A-Za-z0-9!?\']", " ", in)
text = re.sub(r"what's", "what is ", text)
text = re.sub(r"\'s", " ", text)
text = re.sub(r"\'ve", " have ", text)
text = re.sub(r"n't", " not ", text)
text = re.sub(r"i'm", "i am ", text)
text = re.sub(r"\'re", " are ", text)
text = re.sub(r"\'d", " would ", text)
text = re.sub(r"\'ll", " will ", text)
text = re.sub(r"!", " ! ", text)
text = re.sub(r"\?", " ? ", text)
text = re.sub(r"'", " ", text)
out = text.lower()
```

**Figure 6: Basic Data Cleaning in Python Converting *in* to *out*. Here, `re.sub` substitutes any substring in its third argument text matching its first argument regular expression, with its second argument.**

```
out = ''
for word, pos in
  nltk.pos_tag(nltk.word_tokenize(in)):
  if pos == 'NN' or pos == 'NNP' or
     pos == 'NNS' or pos == 'NNPS':
     out = out + ' ' + word.lower()
```

**Figure 7: Noun Removal in Python Converting *in* to *out*. Here we use the NLTK [3] package, where `word_tokenize` splits *in* into words it contains, and `pos_tag` tags each word in the list. Here we include all words tagged by NLTK with either `NN`, `NNP`, `NNS`, or `NNPS`, denoting nouns.**

## 3.2 Data Cleaning

Before using collected data for model training, we need to clean them. We perform the following three steps:

1. The first is the *basic* cleaning procedure. Here, we remove non-alphanumeric characters except exclamation and question marks, and we expand apostrophes. Figure 6 shows the Python procedure.
2. We remove non-noun words that our data collection system can recognize automatically using the NLTK Python package [3] (Figure 7). Using only nouns have been found to be effective for bug assignment recommendation [32]. Our non-noun filtering actually improves model performance, signifying the importance of the focus on nouns in prediction quality (see the experimental results in Section 5.4).
3. We also remove the words which appear in more than 30% of the NVD vulnerability data, since they are common words

[1] $\vec{y}_i$ is not *1-hot* encoded, which requires that there is exactly a single $j$ where $\vec{y}_{ij} = 1$.

which are likely to not help in identifying library names. These are not only stop words, but also include words like "security," which appears in most of the NVD entries. Such words reduce the performance as they are not specific to a particular CVE or set of libraries. This is done using the `CountVectorizer` API of scikit-learn 0.20 [8]. We chose the 30% frequency limit due to its more favorable results when compared with other limits (see Section 5.4).

We show experimentally how our data cleaning approaches improve model performance in Section 5.4.

## 3.3 Feature Engineering and Selection

From each NVD entry we select description, CPE configurations, and references for our model training since the other features are unlikely to help in identifying library names. These other features include management data such as data format, version, timestamp, and dates. They also encompass codified problem type including a *Common Weakness Enumeration* (*CWE*) id, and also impact data which include *Common Vulnerability Scoring System* (*CVSS*) information, severity level, exploitability and impact scores. We clean the description as explained in Section 3.2. The CPE configurations are important features to predict the related library names, as each is made up from vendor name and product name which are very close to library coordinates. There are different versions of CPE formats, but here we consider only the latest version 2.3 at the time of writing. For a given NVD vulnerability entry, there can be multiple CPE configurations. We extract vendor name and product name from each configuration and treat the pair as one unit of text. For example, for the CPE configuration

```
cpe:2.3:a:arastta:ecommerce:1.6.2:*:*:*:*:*:*:*
```

we extract `arastta` (vendor name) and `ecommerce` (product name), and consider them as a text "`arastta ecommerce`." Finally we get a list of "vendor product" for each NVD entry without duplication. There are also multiple reference web links for each NVD entry. From each link, we remove the protocol part (`http`, `https`, or `ftps`) and replace the characters in the set `/=&?` with a white space.

We note that one may be tempted to simply use the "vendor product" pairs for matching with vulnerable libraries. However, there are two problems with this approach:

1. **CPE configurations may not identify all relevant libraries.** There are possibly more relevant libraries than the "vendor product" pairs identified in the CPE configuration, lowering the *recall@k* metric (see Section 4.2).
2. **CPE configurations do not identify the most relevant libraries.** Other libraries that are not specified in the CPE configurations may have higher relevance, lowering the *precision@k* metric (see Section 4.2).

The experimental results of using CPE configurations alone to identify relevant libraries have actually been discouraging, as discussed in Section 5.3.

## 3.4 Matchers

Initially we use the combination of all selected NVD entry features including description, CPE configurations, and references into a single textual feature. Our tests show, however, that for some entries, even though the description or the CPE configurations contain the exact coordinate1 or coordinate2 (see Section 3.1) for the library, the model fails to map the feature vector to the corresponding library name when used in prediction. This led us to use the product name in a CPE configuration to search for all of the matched library coordinates and use these matched coordinates as another input feature. For a library, we call coordinate2 as the *module name* of the library if it has a coordinate2, or we consider coordinate1 to be such. The product name in a CPE configuration is similar or is sometimes the same with the module name. In order to extract more information from the indicated product name, we search all the library entries in the SCA vulnerability database to find those whose module name equals the product name in a CPE configuration. We then add the name ("coordinate1 coordinate2" pair) of this library, called the *matchers*, to our input text data.

Again, a question may arise on the reason of not simply matching CPE configurations with library coordinates. This has been answered in the previous section (Section 3.3), that is, the CPE configurations themselves may not include all relevant libraries, and do not identify the most relevant ones. As we also show in Section 5.3 later, using matcher alone without the rest of the features results in low prediction performance.

In summary, there are four features making up the input used to train the models ($\vec{x}_i$s of Section 3.1), including the cleaned vulnerability description (nouns only), a list of "vendor product" pairs from the CPE configurations, cleaned reference links, and the matchers. We concatenate all features into one contiguous string for each NVD entry. This is how we build the $f_{NVD}$ function of Section 3.1.

# 4 CORE APPROACH

## 4.1 Using FastXML

Consider a domain $\vec{\mathbb{Q}}^{|L|}$ of rational vectors of length $|L|$ (recall that $L$ is the set of labels). For any $\vec{z} \in \vec{\mathbb{Q}}^{|L|}$, $\vec{z}_k$ is called a *score*. The higher the score $\vec{z}_j$, the more "relevant" is a label $L_j$ given $\vec{z}$. The FastXML [28] algorithm produces a *model*, which is a function with signature $\vec{X} \rightarrow \vec{\mathbb{Q}}^{|L|}$ given a finite function with signature $\vec{X} \xrightarrow{\text{fin}} \vec{Y}$, which is the training dataset of Equation 2 of Section 3.1. That is, it is a function:

$$train_{FastXML} : (\vec{X} \xrightarrow{\text{fin}} \vec{Y}) \rightarrow (\vec{X} \rightarrow \vec{\mathbb{Q}}^{|L|}).$$

Given a training dataset $d_{train}$ of Equation 2, the model produced by the FastXML algorithm is $train_{FastXML}(d_{train})$.

Now, our objective is to use the FastXML algorithm for prediction, which in our case is library identification. When $NVD$, $L$, and $\mathbb{Q}$ are respectively the sets of CVE ids, library names, and rational numbers, our library identification function can be formally specified as:

$$identify_{FastXML} : NVD \rightarrow (L \rightarrow \mathbb{Q}).$$

By virtue of the finiteness of $L$, $L \rightarrow \mathbb{Q}$ is finite. We first define a function $\tau : \vec{\mathbb{Q}}^{|L|} \rightarrow (L \rightarrow \mathbb{Q})$ that transforms a finite rational vector of length $|L|$ into the function $L \rightarrow \mathbb{Q}$ as follows:

$$\tau(\vec{r}) = \{(L_j, \vec{r}_j) \mid 1 \leq j \leq |L|\}.$$

Using $\tau$, we define $identify_{FastXML}$ as:

$$identify_{FastXML}(id) = \tau(train_{FastXML}(d_{train})(f_{NVD}(id))).$$

We can sort the elements of the finite $S : L \to \mathbb{Q}$ (the output of $identify_{FastXML}$) in descending order of their right components:

$$sort(S) = (l_1, n_1), \ldots, (l_{|S|}, n_{|S|})$$

where $n_i \geq n_j$ whenever $i < j$. Now, the top-$k$ elements of $S$ is:

$$top_k(S) = \{(l_1, n_1), \ldots, (l_k, n_k)\}$$

when $sort(S) = (l_1, n_1), \ldots, (l_k, n_k), (l_{k+1}, n_{k+1}), \ldots, (l_{|S|}, n_{|S|})$. Given a fixed $k$, we define our implementation as the function:

$$identify_k(id) \quad = \quad top_k \cdot identify_{FastXML}(id)$$

It is easy to see that using $identify_k$ as $identify$ satisfies Equation 1 of Section 1.

## 4.2 Model Evaluation

We use the prediction result to help our security researchers to map CVE ids to library names to save their manual research effort. As usual in evaluating multi-label learning approaches, we use *precision@k*, *recall@k*, and their harmonic mean $F_1@k$ as validation metrics. *precision@k* is the precision of the *top-k* prediction results, where *recall@k* is the recall of the *top-k* prediction results. These metrics focus on the positive labels only, and are therefore suitable for use in XML due to the number of positive labels for an input data item is very small compared to the irrelevant negative labels (less than ten vs. thousands for our typical case). This characteristic renders other methods such as Hamming loss inappropriate [22, 28]. Weston et al. show for the first time the utility of optimizing *precision@k* for an XML application in image recognition [41]. The metrics have also found applications in machine learning approaches for software engineering such as in library recommendation [35, 54], in finding analogical libraries [15], and in tag recommendation [33, 38, 39, 46, 51, 55], essentially where the number of positive labels is magnitudes smaller than the number of negative labels. We define the metrics in this section.

We assume that the results of a manual labeling of CVE id with library names by the security researchers is given by the function:

$$identify_{manual} : NVD \to \mathcal{P}(L)$$

This is indeed the validation dataset. Given a prediction $identify_k(id)$ under the bound $k$, we define the *precision@k* and *recall@k* for a given CVE id as follows:

$$precision@k(id) \quad = \quad \frac{|identify_k(id) \cap identify_{manual}(id)|}{k}$$

$$recall@k(id) \quad = \quad \frac{|identify_k(id) \cap identify_{manual}(id)|}{|identify_{manual}(id)|}$$

*precision@k(id)* here is therefore the proportion of the correctly-predicted names among the maximum $k$ of predicted names for a given id, whereas *recall@k(id)* here is the proportion of the correctly-predicted names among all correct names for a given id. We care about maximizing *precision@k(id)* since we want to save the manual effort of confirming that the CVE id is actually related to the predicted library name. We also care about maximizing *recall@k(id)* since we want our results to cover as many of the related library names as possible.

**Table 1: Dataset Sizes**

| Dataset | No. Entries |
|---|---|
| NVD | 130,115 |
| SCA Library Vulnerability Database | 74,664 |
| SCA Library Data | 2,106,242 |
| Labeled Data | 7,696 |

**Table 2: FastXML Training Parameter Values**

| Parameter | Value |
|---|---|
| Number of Trees | 64 |
| Parallel Jobs | No. of CPUs |
| Max. Leaf Size | 10 |
| Max. Labels per Leaf | 20 |
| Re-Split Count. The number of node re-splitting tries using PfastreXML re-ranking classifier. | 0 |
| Subsampling Data Size. 1 = no subsampling. | 1 |
| Sparse Multiple. Constant for deciding the data structure to use in *nDCG* computation. | 25 |
| Random Number Seed | 2016 |

Given $NVD_v$ the subset of NVD CVE ids that we use for validation, the metrics *precision@k* and *recall@k* that we actually use for validation are as follows:

$$precision@k \quad = \quad \underset{id \,\in\, NVD_v}{\textbf{avg}} \; precision@k(id)$$

$$recall@k \quad = \quad \underset{id \,\in\, NVD_v}{\textbf{avg}} \; recall@k(id)$$

with **avg** denoting arithmetic mean. We use these metrics or their harmonic mean $F_1@k$ to evaluate our models.

# 5 EXPERIMENTS

## 5.1 Research Questions

We experimented with various aspects of our design and report the performance results in this section. We conduct the experiments to answer the following research questions:

RQ1 What is the performance of using matchers alone as input?

RQ2 Does adding description, CPE configurations, and references of NVD entries improve the model performance?

RQ3 Do non-noun and frequent-words removal improve model performance?

RQ4 What is the number of the FastXML trees that results in the best performance?

## 5.2 Dataset and Setup

Table 1 shows the sizes of our input data. Other than the NVD and SCA library vulnerability database, we also retrieve SCA library data, containing data on libraries that may not currently be associated with any CVE id. We use the SCA library data to build the matchers. Our final labeled dataset contains 7,696 records, which is only about 6% of the total number of NVD entries. This is because our SCA databases have a limited focus on open-source projects and a number of supported languages.

Table 3 shows the average and distribution of the number of labels in the labeled dataset. The distribution is skewed, where

**Table 3: Label Number Average and Distribution**

| **Arith. Avg.** | **1** | **2** | **3** | **4** | **≥ 5** |
|---|---|---|---|---|---|
| 4.90 | 60.58% | 20.60% | 5.21% | 3.99% | 9.63% |

**Table 4: Matchers-Only Performance**

| *k* | *precision@k* | *recall@k* | $F_1$*@k* |
|---|---|---|---|
| 1 | 0.52 | 0.37 | 0.43 |
| 2 | 0.39 | 0.48 | 0.43 |
| 3 | 0.30 | 0.52 | 0.38 |
| **Avg.** | 0.39 | 0.45 | 0.41 |

more than 60% of the entries only have one label. This agrees with the sparsity characteristic of the XML problems. We conduct all experiments on Amazon EC2 instance running Ubuntu 18.04 with 32 GB RAM and 16-core 3 GHz Intel(R) Xeon(R) Platinum 8124M CPU. We use the default parameters for FastXML excluding two:

- We changed the number of trees from the default 1 to 64.
- We changed the number of parallel jobs to the number of detected CPUs (16).

Table 2 summarizes the FastXML parameter values for parameters that affect our experiments. The FastXML implementation that we use has other parameters (see the constructor of `Trainer` class in the source code [2]). They are for training classifiers for PfastreXML node re-splitting, however, since we do not use this feature (in Table 2 we set the re-split count to 0), we do not list them in Table 2. We also exclude parameters that are not actually used and those that are only used for reporting purposes.

Our standard approach, unless indicated otherwise, is to randomly select 75% of our labeled dataset to train the models and the remaining 25% for testing. When presenting performance results, we use geometric average unless indicated otherwise [19]. Geometric average is important to present the performance improvements in Section 6. We note that our problem is time-agnostic, where the identification of libraries are not affected by the timestamps of the NVD entries. Our dataset is therefore not sorted based on time, and this removes the necessity of using sliding window-based validation techniques.

## 5.3 Using Matchers Only

We note that matchers (see Section 3.4) alone can be used for prediction. We perform an experiment to determine the performance when only matchers are included in the input. The results are shown in Table 4. **It is easy to see the answer to RQ1, that the prediction performances are low when only matchers are included in the input, with average $F_1$@*k* of only 0.41.**

## 5.4 Experiments with Data Cleaning

We perform experiments to measure the effect of our data preparation approaches to the model performance. Table 5 shows the the *precision@k*, *recall@k*, $F_1$*@k* results for $k$ = 1 and 3, and the training and validation times (using the 75% and 25% labeled data). When we compare Table 5, with Table 4, **we can answer RQ2 in the affirmative, that the addition of description, CPE configurations, and references of NVD entries does improve the model performance**. Comparing the minimum 0.50 average $F_1$*@k* of Table 5 and the average $F_1$*@k* of 0.41 of Table 4, we get the minimum improvement in average $F_1$*@k* to be 21.95%. The best performance in Table 5 is for the configuration with non-noun and 30% frequent words removal. We use this configuration in production. **Here we answer RQ3 in the affirmative: non-noun removal and frequent-words removal improve the prediction performance.** Although we can observe that the difference in average $F_1$*@k* score for our production configuration compared to others is not much, non-noun and frequent-words removal still reduce training and validation times.

**Table 5: *Precision@k*, *Recall@k*, and $F_1$*@k* Results ($k$ = 1, 2, and 3). T = training time, V = validation time. The configuration that we use in production is of the shaded column.**

| | | **Basic Cleaning** | | | | |
|---|---|---|---|---|---|---|
| | | | **Non-Noun Removal** | | | |
| | | | | **Frequent-Words Removal** | | |
| | *k* | | | **30%** | **60%** | **90%** |
| *pre.@k* | 1 | 0.64 | 0.65 | 0.65 | 0.65 | 0.65 |
| | 2 | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 |
| | 3 | 0.37 | 0.37 | 0.37 | 0.37 | 0.37 |
| | **Avg.** | 0.48 | 0.49 | 0.49 | 0.49 | 0.49 |
| *rec.@k* | 1 | 0.44 | 0.45 | 0.45 | 0.45 | 0.45 |
| | 2 | 0.57 | 0.58 | 0.58 | 0.58 | 0.58 |
| | 3 | 0.61 | 0.61 | 0.62 | 0.61 | 0.61 |
| | **Avg.** | 0.53 | 0.54 | 0.54 | 0.54 | 0.54 |
| $F_1$*@k* | 1 | 0.52 | 0.53 | 0.53 | 0.53 | 0.53 |
| | 2 | 0.52 | 0.53 | 0.53 | 0.53 | 0.53 |
| | 3 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 |
| | **Avg.** | 0.50 | 0.50 | 0.51 | 0.50 | 0.50 |
| **T (S)** | | 175 | 161 | 159 | 160 | 160 |
| **V (S)** | | 219 | 200 | 197 | 199 | 200 |

**Table 6: *Precision@k*, *Recall@k*, $F_1$*@k* ($k$ = 1,2,3) and Training (T) and Validation (V) Times with Various Tree Numbers. The tree number that we use is 64 (shaded columns).**

| | | **No. Trees** | | | | **No. Trees** | | |
|---|---|---|---|---|---|---|---|---|
| | *k* | 32 | 64 | 128 | | 32 | 64 | 128 |
| *pre.@k* | 1 | 0.64 | 0.65 | 0.66 | *rec.@k* | 0.45 | 0.45 | 0.46 |
| | 2 | 0.47 | 0.48 | 0.48 | | 0.57 | 0.58 | 0.58 |
| | 3 | 0.36 | 0.37 | 0.37 | | 0.61 | 0.62 | 0.62 |
| | **Avg.** | 0.48 | 0.49 | 0.49 | | 0.54 | 0.54 | 0.55 |
| $F_1$*@k* | 1 | 0.53 | 0.53 | 0.54 | T (S) | 80 | 219 | 317 |
| | 2 | 0.52 | 0.53 | 0.53 | V (S) | 108 | 227 | 375 |
| | 3 | 0.45 | 0.46 | 0.46 | | | | |
| | **Avg.** | 0.50 | 0.51 | 0.51 | | | | |

## 5.5 Experiments with Tree Sizes

FastXML uses trees, where each one represents a distinct hierarchy over the feature space. We perform an experiment which varies the number of trees among 32, 64, and 128, and summarize the results in Table 6. We also include the time measurement results, both for training and validation, on respectively the 75% and 25% of the labeled data. For RQ4, we confirm that **the number of FastXML trees that result in the best performance is 128.** However, the

**Table 7: Dataset Sizes for Production Model Training and Validation**

| Dataset | No. Entries |
|---|---|
| NVD | 126,219 |
| SCA Library Vulnerability Database | 73,224 |
| SCA Library Data | 2,030,578 |
| Labeled Data | 7,153 |

**Table 8: Label Number Average and Distribution for Production Model Training and Validation Labeled Dataset**

| Arith. Avg. | 1 | 2 | 3 | 4 | ≥ 5 |
|---|---|---|---|---|---|
| 5.09 | 59.95% | 20.58% | 5.77% | 3.96% | 9.74% |

difference in average $F_1@k$ between 64 and 128 is very small, and since the 64 trees configuration requires only half the training time, we use it for our production system.

# 6 DEPLOYMENT CASE STUDY

We want to confirm that a newly-trained model has a better performance than the model in production before replacing the production model. In this section we discuss the two steps of validation that we perform for this purpose via a case study. In this case study, for the data preparation we use basic cleaning, non-noun removal, and with 30% frequent words removal. We add matchers into the input. For the number of FastXML trees, we use 64. We use the same system setup as the one mentioned in Section 5.2.

## 6.1 Evaluation Using Training-Time Datasets

The first step is to ensure that the new model has a better performance than the production model, at the moment each one is trained. For our case study, the production model is trained on 18 July 2019 and the new model is trained on 27 September 2019 (70 days difference). In training and validating the production model, the sizes of the datasets that we use are shown in Table 7. For the new model, the sizes of the datasets that we use are presented in Table 1. We note that each dataset of Table 7 is a subset of its counterpart in Table 1. We show the label number average and distribution for the labeled dataset of Table 7 in Table 8. We observe that the numbers are characteristically similar to that of Table 3.

For this step, we use 75% of the data for training and 25% for testing, for both the production and the new models. As most of the NVD entries are associated with one or two labels, we evaluate the top-$k$ labels from the prediction results, for $k$ = 1, 2, or 3. Columns 3–5 of Table 9 show the results for the production and new models. Although the new model shows a decrease in precision, the average $F_1@k$ still improves by 1.09%, hence by this evaluation step, deploying the new model as a replacement for the production model is still acceptable.

## 6.2 Evaluation Using the Same Dataset

For the second evaluation, we randomly select 50% of the labeled data from the labeled dataset of the time we train the new model. We build this dataset from 25% labeled data used for testing the new model and a third of the remaining 75% labeled data used for training the new model. We test the new model and the production model on this same dataset. Columns 6–7 of Table 9 show the performance comparison between the production and the new models. All metrics *precision@k*, *recall@k*, and $F_1@k$ show significant improvements by the new model, providing us with more confidence in deploying the new model. We note that in this second evaluation, the performance numbers are higher when compared to the evaluating using training-time datasets. This is because half of the data in the dataset for the same-dataset evaluation are used in the training of the new model as well.

**Table 9: *Precision@k*, *Recall@k*, and $F_1@k$ Results ($k$ = 1, 2, and 3) for the Deployment Case Study. I% = percent improvement of the new model vs. the production.**

| | | Training-Time Datasets | | | Same Dataset | | |
|---|---|---|---|---|---|---|---|
| | $k$ | Prod. | New | I% | Prod. | New | I% |
| *pre@k* | 1 | 0.63 | 0.65 | 3.17 | 0.75 | 0.94 | 25.33 |
| | 2 | 0.49 | 0.48 | -2.04 | 0.56 | 0.68 | 21.42 |
| | 3 | 0.38 | 0.37 | -2.63 | 0.44 | 0.52 | 18.18 |
| | Avg. | 0.49 | 0.49 | -0.53 | 0.57 | 0.69 | 21.61 |
| *rec@k* | 1 | 0.43 | 0.45 | 4.65 | 0.52 | 0.68 | 30.77 |
| | 2 | 0.57 | 0.58 | 1.75 | 0.67 | 0.85 | 26.87 |
| | 3 | 0.60 | 0.62 | 3.33 | 0.72 | 0.89 | 23.61 |
| | Avg. | 0.53 | 0.54 | 3.24 | 0.63 | 0.80 | 27.05 |
| $F_1@k$ | 1 | 0.51 | 0.53 | 4.05 | 0.61 | 0.79 | 28.49 |
| | 2 | 0.53 | 0.53 | -0.32 | 0.61 | 0.76 | 23.85 |
| | 3 | 0.47 | 0.46 | -0.40 | 0.55 | 0.66 | 20.18 |
| | Avg. | 0.50 | 0.51 | 1.09 | 0.59 | 0.73 | 24.13 |

**Table 10: Training and Prediction Times in Seconds. The prediction time is on the same dataset, which encompasses 50% of the labeled data for training and validation.**

| | Prod. | New | Arithmetic Average |
|---|---|---|---|
| Training | 147 | 159 | 153 |
| Prediction | 62 | 64 | 63 |

## 6.3 Training and Prediction Times

Table 10 shows the training and prediction times of our production and new models and their averages. The prediction times are from the evaluation using the same dataset (see Section 6.2), which is 50% of all the labeled dataset used in the training and validation of the new model. Table 10 shows that both the training and prediction times are fast, roughly about 2.5 minutes and 1 minute, respectively. This means that prediction finishes on each data item on average in 8.17 ms. This demonstrates that our approach is highly practical.

# 7 THREATS TO VALIDITY

## 7.1 Internal Threats to Validity

We identify two kinds of internal threats of validity. Firstly, our results are exposed to human error from our manually-built data sources, including NVD and our SCA vulnerable library database. It is possible that the SCA vulnerability library identifies more or less library as related to an NVD entry than it should. This threat is mitigated by the updates to the NVD by the security community and the proven commercial usage of the SCA vulnerable library database. Secondly, our results may also be affected by the possible bugs and errors in our implementation. This is partly mitigated by using widely-used standard library packages for machine learning, including pandas 0.23.4, scikit-learn 0.20, and FastXML 2.0.

### 7.2 External Threats to Validity

There are two threats to the generalizability of our study. Firstly, as mentioned, SCA vulnerable library database is curated manually, from which we build our input feature vector. This manual curation is highly dependent on the skills of the security researchers. Secondly, we are only interested in the NVD entries that are related with open source projects and whose languages are supported by the Veracode SCA, so the final size of the labeled dataset for producing the models is only about 6% of the total number of NVD entries (see Tables 1 and 7).

## 8 RELATED WORK

### 8.1 Machine Learning for SCA

Our work is part of a framework [47] for software composition analysis (SCA), which has gained widespread industrial acceptance. Machine learning is extensively used in SCA to identify vulnerable libraries, which are vulnerabilities that are known by some (e.g., developers making the vulnerability fix), yet not explicitly declared, such as via the NVD. Wijayasekara et al. [43] point to the criticality of such vulnerabilities, whose number has increased in Linux kernel and MySQL by 53% and 10% respectively from 2006 to 2011. Zhou and Sharma [56] explore the identification of vulnerabilities in commit messages and issue reports/PRs using machine learning. Their approach discovers hidden vulnerabilities in more than 5,000 projects spanning over six programming languages. Sabetta and Bezzi propose feature extraction also from commit patches in addition to commit messages [30]. Wan considers vulnerability identification from commit messages using deep learning [37]. Compared to these, our work solves a different problem, that of the vulnerable library identification from NVD data.

### 8.2 Library Recommendation

Close to our work is the area of library recommendation. Thung et al. propose an approach that recommends other third-party libraries to use, given the set of third-party libraries currently used by the application or similar applications [35]. The approach uses a combination of association rule mining and collaborative filtering. Zhao et al. propose an improvement using the application's description text features and the text features obtained from the libraries themselves, using NLP technique and collaborative filtering [54]. Compared to Thung et al. which relies on the Maven dependency information, this makes Zhao et al.'s approach language agnostic. Chen et al. considers a related problem, which is that of finding *analogical* libraries [15, 16]. Here, the problem is in discovering libraries that are related to a target library. To build this relation, they treat the tags of StackOverflow question (which may include library names) as a sentence and apply word embedding. The solutions to library recommendation problem take advantage of the intuition that applications can be categorized according to the libraries they use. Both the library recommendation and analogous libraries problems can be considered as XML instances. However, the library recommendation problem is amenable to methods that narrow down the possible recommendations. This is less applicable in our setting since the correlation between vulnerabilities and the set of libraries having them is weaker, or even possibly nonexistent. Compared to the analogous libraries problem, in our case CPE configurations can be considered as tags, however, we cannot take them at face value to identify libraries: We need to also identify libraries not mentioned in the CPE configurations.

### 8.3 Multi-Label Classification for Software Engineering

Multi-label (ML) classification has found many uses in the area of software engineering. Prana et al. proposes eight-label classifier to categorize the sections of Github README files [29]. The solution combines BR with *support-vector machine* (*SVM*) as the base classifier. It achieves $F_1$ score of 0.746. Feng and Chen maps execution traces to types of faults using ML-KNN [52] ML algorithm [17]. Xia et al. improves Feng and Chen's results using MLL-GA [21] algorithm instead [45]. Feng et al. provide a comparison of various ML algorithms [18]. Xia et al. propose *TagCombine*, which models tag recommendation in software information sites such as StackOverflow as ML classification problem [46]. Short et al. confirm that by adding information about the network of the StackOverflow posts, results can be improved [33]. *EnTagRec* [38] and subsequently *EnTagRec++* [39] improve TagCombine by using a mixture model that considers all tags together instead of building one classifier for each tag (BR). Zhou et al. propose a scalable solution that considers only a subset of posts data to build the recommendation [55]. Their approach also improves $F_1$@10 score by 8.05% when compared to EnTagRec. Zavou applies deep learning to the tag recommendation problem and demonstrates improvements over TagCombine for AskUbuntu data [51]. *SOTagger* approach considers the related problem of tagging the posts using intent rather than the technology [36]. Our problem cannot be categorized as multi-label classification problem, as we consider thousands of libraries (labels) in total. In particular, the usual ML approach using BR is not applicable.

## 9 CONCLUSION AND FUTURE WORK

Predicting related libraries for NVD CVE entries is an important step in SCA product to save manual research effort in the identification of vulnerable libraries. In this article, we present the design and implementation of a system that performs data collection, feature engineering, model training, validation and prediction automatically, and its experimental evaluations. At the time of writing, our system achieves $F_1$@1 score of 0.53 with average $F_1$@$k$ score for $k = 1, 2, 3$ of 0.51 ($F_1$@$k$ is the harmonic mean of *precision@k* and *recall@k*, all of which are defined in Section 4.2). Our problem, which is that of mapping NVD entries to library names is an instance of XML. We base our system on the tree-based FastXML [28] approach, which solves XML. At present, we do not employ deep learning as our labeled data size is currently not big enough. As a future work, one can try deep learning methods if more data available.

## ACKNOWLEDGMENT

We thank Jonah De La Cruz for providing Figure 4 and Abhishek Sharma for commenting on our draft. The last author is supported by Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant (MOE2019-T2-1-193).

## REFERENCES

[1] [n.d.]. Black Duck Software Composition Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.

[2] [n.d.]. FastXML / PFastXML / PFastreXML - Fast and Accurate Tree Extreme Multi-label Classifier. https://github.com/refefer/fastxml.

[3] [n.d.]. Natural Language Toolkit. https://www.nltk.org/.

[4] [n.d.]. NVD - CVE-2011-0448. https://nvd.nist.gov/vuln/detail/CVE-2011-0448.

[5] [n.d.]. NVD - CVE-2015-7318. https://nvd.nist.gov/vuln/detail/CVE-2015-7318.

[6] [n.d.]. NVD - Data Feeds. https://nvd.nist.gov/vuln/data-feeds.

[7] [n.d.]. NVD - Home. https://nvd.nist.gov/.

[8] [n.d.]. scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/.

[9] [n.d.]. Software Composition Analysis. https://www.flexera.com/products/software-composition-analysis.

[10] [n.d.]. Software Composition Analysis — Veracode. https://www.veracode.com/products/software-composition-analysis.

[11] [n.d.]. Vulnerability Scanner. https://www.sonatype.com/appscan.

[12] R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. 2013. Multi-label learning with millions of labels: recommending advertiser bid phrases for web pages. In *22nd WWW*. Intl. WWW Conferences Steering Committee / ACM, 13–24.

[13] R. Babbar and B. Schölkopf. 2017. DiSMEC: Distributed Sparse Machines for Extreme Multi-label Classification. In *10th WSDM*. ACM, 721–729.

[14] K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain. 2015. Sparse Local Embeddings for Extreme Multi-label Classification. In *28th NIPS*. 730–738.

[15] C. Chen, S. Gao, and Z. Xing. 2016. Mining Analogical Libraries in Q&A Discussions - Incorporating Relational and Categorical Knowledge into Word Embedding. In *23rd SANER*. IEEE Comp. Soc., 338–348.

[16] C. Chen and Z. Xing. 2016. SimilarTech: automatically recommend analogical libraries across different programming languages. In *31st ASE*. IEEE Comp. Soc., 834–839.

[17] Y. Feng and Z. Chen. 2012. Multi-label software behavior learning. In *34th ICSE*. IEEE Comp. Soc., 1305–1308.

[18] Y. Feng, J. A. Jones, Z. Chen, and C. Fang. 2018. An Empirical Study on Software Failure Classification with Multi-label and Problem-Transformation Techniques. In *11th ICST*. IEEE Comp. Soc., 320–330.

[19] P. J. Fleming and J. J. Wallace. 1986. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Commun. ACM* 29, 3 (1986), 218–221.

[20] D. Foo, J. Yeo, X. Hao, and A. Sharma. 2019. The Dynamics of Software Composition Analysis. *CoRR* abs/1909.00973 (2019). arXiv:1909.00973

[21] D. E. Goldberg and J. H. Holland. 1988. Genetic Algorithms and Machine Learning. *Machine Learning* 3 (1988), 95–99.

[22] H. Jain, Y. Prabhu, and M. Varma. 2016. Extreme Multi-label Loss Functions for Recommendation, Tagging, Ranking & Other Missing Label Applications. In *22nd KDD*. ACM, 935–944.

[23] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *CoRR* abs/1607.01759 (2016).

[24] J. Liu, W.-C. Chang, Y. Wu, and Y. Yang. 2017. Deep Learning for Extreme Multi-label Text Classification. In *40th SIGIR*. ACM, 115–124.

[25] T.-Y. Liu. 2011. *Learning to Rank for Information Retrieval.*

[26] O. Luaces, J. Díez, J. Barranquero, J. J. del Coz, and A. Bahamonde. 2012. Binary relevance efficacy for multilabel classification. *Progress in AI* 1, 4 (2012), 303–313.

[27] I. Partalas, A. Kosmopoulos, N. Baskiotis, T. Artières, G. Paliouras, É. Gaussier, I. Androutsopoulos, M.-R. Amini, and P. Gallinari. 2015. LSHTC: A Benchmark for Large-Scale Text Classification. *CoRR* abs/1503.08581 (2015).

[28] Y. Prabhu and M. Varma. 2014. FastXML: a fast, accurate and stable tree-classifier for extreme multi-label learning. In *20th KDD*. ACM, 263–272.

[29] G. A. A. Prana, C. Treude, F. Thung, T. Atapattu, and D. Lo. 2019. Categorizing the Content of GitHub README Files. *Empirical Software Engineering* 24, 3 (2019), 1296–1327.

[30] A. Sabetta and M. Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *34th ICSME*. IEEE Comp. Soc.

[31] K. Shah, S. Köprü, and J.-D. Ruvini. 2018. Neural Network based Extreme Classification and Similarity Models for Product Matching. In *NAACL-HLT '18*. ACL, 8–15.

[32] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *10th MSR*. IEEE Comp. Soc., 2–11.

[33] L. Short, C. Wong, and D. Zeng. 2014. *Tag Recommendations in StackOverflow*. Technical Report. Stanford University. CS224W Final Project.

[34] Y. Tagami. 2017. AnnexML: Approximate Nearest Neighbor Search for Extreme Multi-label Classification. In *23rd KDD*. ACM, 455–464.

[35] F. Thung, D. Lo, and J. L. Lawall. 2013. Automated library recommendation. In *20th WCRE*. IEEE Comp. Soc., 182–191.

[36] A. S. M. Venigalla, C. S. Lakkundi, and S. Chimalakonda. 2019. SOTagger - Towards Classifying Stack Overflow Posts through Contextual Tagging (S). In *31st SEKE*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 493–639.

[37] L. Wan. 2019. *Automated Vulnerability Detection System Based on Commit Messages*. Master's thesis. Nanyang Technological University.

[38] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik. 2014. EnTagRec: An Enhanced Tag Recommendation System for Software Information Sites. In *30th ICSM*. IEEE Comp. Soc., 291–300.

[39] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik. 2018. EnTagRec ++: An enhanced tag recommendation system for software information sites. *Empirical Software Engineering* 23, 2 (2018), 800–832.

[40] Y. Wang, L. Wang, Y. Li, D. He, and T.-Y. Liu. 2013. A Theoretical Analysis of NDCG Type Ranking Measures. In *26th COLT (JMLR Workshop and Conf. Proceedings)*, Vol. 30. JMLR.org, 25–54.

[41] J. Weston, S. Bengio, and N. Usunier. 2011. WSABIE: Scaling Up to Large Vocabulary Image Annotation. In *22nd IJCAI*. IJCAI/AAAI, 2764–2770.

[42] J. Weston, A. Makadia, and H. Yee. 2013. Label Partitioning For Sublinear Ranking. In *30th ICML (JMLR Workshop and Conf. Proceedings)*, Vol. 28. JMLR.org, 181–189.

[43] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *5th HSI*. IEEE, 89–96.

[44] T.-F. Wu, C.-J. Lin, and R. C. Weng. 2004. Probability Estimates for Multi-class Classification by Pairwise Coupling. *J. Mach. Learn. Res.* 5 (2004), 975–1005.

[45] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang. 2014. Towards more accurate multi-label software behavior learning. In *CSMR-WCRE '14*. IEEE Comp. Soc., 134–143.

[46] X. Xia, D. Lo, X. Wang, and B. Zhou. 2013. Tag recommendation in software information sites. In *10th MSR*. IEEE Comp. Soc., 287–296.

[47] C. Yang, A. E. Santosa, M. Y. Ang, A. Sharma, and D. Lo. [n.d.]. A Machine Learning Approach for Vulnerability Curation. ([n. d.]). Submitted to ICSE '20 SEIP.

[48] I. E.-H. Yen, X. Huang, W. Dai, P. Ravikumar, I. S. Dhillon, and E. P. Xing. 2017. PPDsparse: A Parallel Primal-Dual Sparse Method for Extreme Classification. In *23rd KDD*. ACM, 545–553.

[49] I. E.-H. Yen, X. Huang, P. Ravikumar, K. Zhong, and I. S. Dhillon. 2016. PD-Sparse: A Primal and Dual Sparse Approach to Extreme Multiclass and Multilabel Classification. In *33rd ICML (JMLR Workshop and Conf. Proceedings)*, Vol. 48. JMLR.org, 3069–3077.

[50] R. You, S. Dai, Z. Zhang, H. Mamitsuka, and S. Zhu. 2018. AttentionXML: Extreme Multi-Label Text Classification with Multi-Label Attention Based Recurrent Neural Networks. *CoRR* abs/1811.01727 (2018). arXiv:1811.01727

[51] C. Zavou. 2017. *Question Retrieval in Community Question Answering Enhanced by Tags Information in a Deep Neural Network Framework*. Master's thesis. University of Amsterdam.

[52] M.-L. Zhang and Z.-H. Zhou. 2007. ML-KNN: A lazy learning approach to multi-label learning. *Pattern Recognition* 40, 7 (2007), 2038–2048.

[53] W. Zhang, J. Yan, X. Wang, and H. Zha. 2018. Deep Extreme Multi-label Learning. In *ICMR '18*. ACM, 100–107.

[54] X. Zhao, S. Li, H. Yu, Y. Wang, and W. Qiu. 2019. Accurate Library Recommendation Using Combining Collaborative Filtering and Topic Model for Mobile Development. *IEICE Transactions* 102-D, 3 (2019), 522–536.

[55] P. Zhou, J. Liu, Z. Yang, and G. Zhou. 2017. Scalable tag recommendation for software information sites. In *24th SANER*. IEEE Comp. Soc., 272–282.

[56] Y. Zhou and A. Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *11th FSE*. ACM, 914–919.