Automatic feature learning for vulnerability prediction

Hoa Khanh Dam*, Truyen Tran†, Trang Pham†, Shien Wee Ng*, John Grundy† and Aditya Ghose*
*University of Wollongong, Australia
Email: {hoa,swn881,aditya}@uow.edu.au
†Deakin University, Australia
Email: {truyen.tran,phtra,j.grundy}@deakin.edu.au

***Abstract*—Code flaws or vulnerabilities are prevalent in software systems and can potentially cause a variety of problems including deadlock, information loss, or system failure. A variety of approaches have been developed to try and detect the most likely locations of such code vulnerabilities in large code bases. Most of them rely on manually designing features (e.g. complexity metrics or frequencies of code tokens) that represent the characteristics of the code. However, all suffer from challenges in sufficiently capturing both semantic and syntactic representation of source code, an important capability for building accurate prediction models. In this paper, we describe a new approach, built upon the powerful deep learning Long Short Term Memory model, to automatically learn both semantic and syntactic features in code. Our evaluation on 18 Android applications demonstrates that the prediction power obtained from our learned features is equal or even superior to what is achieved by state of the art vulnerability prediction models: 3%–58% improvement for within-project prediction and 85% for cross-project prediction.**

I. Introduction

A software vulnerability – a security flaw, glitch, or weakness found in software systems – can potentially cause significant damages to businesses and people's lives, especially with the increasing reliance on software in all areas of our society. For instance, the Heartbleed vulnerability in OpenSSL exposed in 2014 has affected billions of Internet users [1]. Cyberattacks are constant threats to businesses, governments and consumers. The rate and cost of a cyber breach is increasing rapidly with annual cost to the global economy from cybercrime being estimated at $400 billion [2]. In 2017, it is estimated that the global security market is worth $120 billion [3]. Central to security protection is the ability to detect and mitigate software vulnerabilities early, especially before software release to effectively prevent attackers from exploit them.

Software has significantly increased in both size and complexity. Identifying security vulnerabilities in software code is highly difficult since they are rare compared to other types of software defects. For example, the infamous Heartbleed vulnerability was caused only by two missing lines of code [4]. Finding software vulnerabilities is commonly referred to as "searching for a needle in a haystack" [5]. Static analysis tools have been routinely used as part of the security testing process but they commonly generate a large number of false positives [6, 7]. Dynamic analysis tools rely on detailed monitoring of run-time properties including log files and memory, and require a wide range of representative test cases to exercise the application. Hence, standard practice still relies heavily on domain knowledge to identify the most vulnerable part of a software system for intensive security inspection.

Software engineers can be supported by automated tools that explore remaining parts of the code base more likely to contain vulnerabilities and raise an alert on these. Such predictive models and tools can help prioritize effort and optimize inspection and testing costs. They aim to increase the likelihood of finding vulnerabilities and reduce the time required by software engineers to discover vulnerabilities. In addition, a predictive capability that identifies vulnerable components early in the software lifecycle is a significant achievement since the cost of finding and fixing errors increases dramatically as the software lifecycle progresses.

A common approach to build vulnerability prediction models is by using machine learning techniques. A number of features representing software code are selected for use as predictors for vulnerability. The most commonly used features in previous work (e.g. [8]) are software metrics (e.g. size of code, number of dependencies, and cyclomatic complexity), code churn metrics (e.g. the number of code lines changed), and developer activity. Those features cannot however distinguish code regions of different semantics. In many cases, two pieces of code may have the same complexity metrics but they behave differently and and thus have different likelihood of vulnerability to attack. Furthermore, the choice of which features are selected as predictors is *manually* chosen by knowledgeable domain experts, and may thus carry outdated experience and underlying biases. In addition, in many situations handcrafted features normally do not generalize well: features that work well in a certain software project may not perform well in other projects [9].

An emerging approach is treating software code as a form of text and leveraging Natural Language Processing (NLP) techniques to automatically extract features. Previous work (e.g. [10]) has used Bag-of-Words (BoW) to represent a source code file as a collection of code tokens associated with frequencies. The terms are the features which are used as the predictors for their vulnerability prediction model. Their set of features are thus not fixed or pre-determined (as seen in the software metric model), but rather depend on the vocabulary used by developers. However, the BoW approach has two major weaknesses. Firstly, it ignores the semantics of code

tokens, e.g. fails to recognize the semantic relations between "for" and "while". Secondly, a bag of code tokens does not necessarily capture the semantic structure of code, especially its sequential nature.

The recent advances of deep learning models [11] in machine learning offer a powerful alternative to software metrics and BoW in representing software code. One of the most widely-used deep learning models is Long Short-Term Memory (LSTM) [12], a special kind of recurrent neural networks that is highly effective in learning long-term dependencies in sequential data such as text and speech. LSTMs have demonstrated ground-breaking performance in many applications such as machine translation, video analysis, and speed recognition [11] .

This paper presents a novel deep learning-based approach to *automatically learn features* for predicting vulnerabilities in software code. We leverage LSTM to capture the long context relationships in source code where dependent code elements are scattered far apart. For example, pairs of code tokens that are required to appear together due to programming language specification (e.g. try and $catch$ in Java) or due to API usage specification (e.g. $lock()$ and $unlock()$), but do not immediately follow each other. The learned features sufficiently represent both the semantics of code tokens (*semantic features*) and the sequential structure of source code (*syntactic features*). Our automatic feature learning approach eliminates the need for manual feature engineering which occupies most of the effort in traditional approaches. Results from our experiments on 18 Java applications for the Android OS platform from a public dataset [10] demonstrate that our approach is highly effective in predicting vulnerabilities in code.

The outline of this paper is as follows. In the next section, we provide a motivation example. Section III provides a brief background on vulnerability prediction and the neural networks used in our model. We then present our approach in Section IV, its implementation in Section V, report the experiments to evaluate it in Section VI, and discuss the threats to validity in Section VII. In Section VIII, we discuss related work before summarizing the contributions of the paper and outlines future work in Section IX.

II. Motivation

Figure 1 shows two code listings in Java which was adapted from [13]. Both pieces of code aim to avoid data corruption in multi-threaded Java programs by protecting shared data from concurrent modifications and accesses (e.g. file f this example). They do so by using a reentrant mutual exclusion lock l to enforce exclusive access to the file f. Here, a thread executing this code means to acquire the lock before reading file f, and then release the lock when it is done with the file to allow other threads to access the file.

The use of such locking can however result in deadlocks. Listing 1 in Figure 1 demonstrates an example of deadlock vulnerabilities. While it reads file f, an exception (e.g. file not found) may occur and control transfers to the $catch$ block. Hence, the call to $unlock()$ never gets executed, and thus it

```
1  try {
2    l.lock()
3    readFile(f);
4    l.unlock();
5  }
6  catch (Exception e) {
7    // Do something
8  }
9  finally {
10   closeFile(f);
11 }
```

Listing 1: File1.java

```
1  l.lock()
2  try {
3    readFile(f);
4  }
5  catch (Exception e) {
6    // Do something
7  }
8  finally {
9    l.unlock();
10   closeFile(f);
11 }
```

Listing 2: File2.java

Fig. 1: A motivating example

fails to release the lock. An unreleased lock in a thread will prevent other threads from acquiring the same lock, leading to a deadlock situation. Deadlock is a serious vulnerability, which can be exploited by attackers to organise Denial of Service (DoS) attacks. This type of attack can slow or prevent legitimate users from accessing a software system.

Listing 2 in Figure 1 rectifies this vulnerability. It fixes the problem of the lock not being released by calling $unlock()$ in the $finally$ block. Hence, it guarantees that the lock is released regardless of whether or not an exception occurs. In addition, the code ensures that the lock is held when the $finally$ block executes by acquiring the lock (calling $lock()$) immediately before the try block.

The two code listings are identical with respect to both software metric and Bag-of-Words measures used by most current predictive and machine learning approaches. The number of code lines, the number of conditions, variables, and branches are the same in both listings. The code tokens and their frequencies are also identical in both pieces of code. Hence, the two code listings are indistinguishable if either software metrics or BoW are used as features. Existing work which relies on those features would fail to recognize that the left-hand side listing contains a vulnerability while the right-hand side does not.

III. Background

A. Vulnerability prediction

Vulnerability prediction typically involves determining whether a source code file is likely to be vulnerable. The goal is to alert software engineers with parts of the code base that deserve particular attention, rather than pinpointing exactly the code line where a vulnerability is resided. Hence, we also choose to work at the level of files since this is also the scope of existing work (e.g. [8, 10]) which we would like to compare our approach against.

Determining if a source file is vulnerable can be considered as a function $vuln(x)$ which takes as input a file x and returns a boolean value: $true$ indicates that the file is vulnerable, while $false$ indicates that the file is clean. Machine learning techniques have been widely used to learn function $vuln(x)$. To make it mathematically and computationally convenient for

machine learning algorithms, file x needs to be represented as a n-dimensional vector where each dimension represents a feature (or predictor).

B. Recurrent Neural Network and Long Short Term Memory

A Recurrent Neural Network (RNN) [14] is a single-hidden-layer neural network repeated multiple times. While a feedforward neural network maps an input vector into an output vector, an RNN maps **a sequence into a sequence** (see Figure 2). Let $\boldsymbol{w}_1, ..., \boldsymbol{w}_n$ be the input sequence (e.g. code tokens) and $\boldsymbol{y}_1, ..., \boldsymbol{y}_n$ be the sequence of corresponding labels (e.g. the next code tokens). At each step t, a standard RNN model reads the input $\boldsymbol{w}_t$ and the previous output state $\boldsymbol{s}_{t-1}$ to compute the output state $\boldsymbol{s}_t$ as follows.

$$\boldsymbol{s}_t = \sigma\left(\boldsymbol{b} + W_{tran}\boldsymbol{s}_{t-1} + W_{in}\boldsymbol{w}_t\right) \quad (1)$$

where σ is a nonlinear element-wise transform function, and $\boldsymbol{b}$, W_{tran} and W_{in} are referred to as *model parameters*. The output state is used to predict the output (e.g. the next code token based on the previous ones) at each step t.

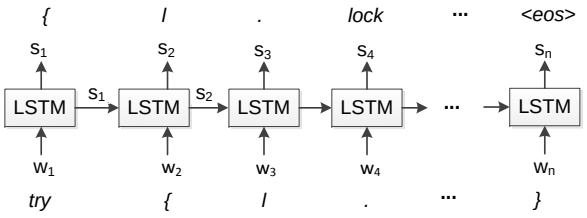

Fig. 2: A recurrent neural network

An RNN shares the same parameters across all steps since the same task is performed at each step, just with different inputs. Hence, using an RNN significantly reduces the total number of model parameters which we need to learn. The RNN model is trained using many input sequences with known true output sequences. The errors between the true outputs and the predicted outputs are passed backwards through the network during training to adjust the model parameters such that the errors are minimized.

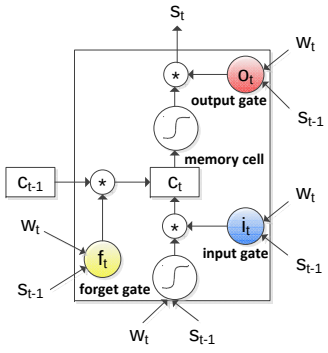

Fig. 3: The internal structure of an LSTM unit

A Long Short-Term Memory (LSTM) [12, 15] architecture is a special variant of RNN, which is capable of learning long-term dependencies. Due to space limitations, we briefly describe LSTM here and refer the readers to the seminal paper [12] for more details. It has a *memory cell* c_t which stores accumulated memory of the context. The amount of information flowing through the memory cell is controlled by three gates (an *input gate* i_t, a *forget gate* f_t, and an *output gate* o_t), each of which returns a value between 0 (i.e. complete blockage) and 1 (full passing through). All those gates are *learnable*, i.e. being trained with the whole code corpus. It is important to note here that LSTM computes the output state based on not just only the current input w_t and the previous ouput state h_{t-1} (as done in standard RNNs) but also the current memory cell state c_t, which is *linear* with the previous memory c_{t-1}. This is the key feature allowing an LSTM model to learn long-term dependencies.

IV. Approach

A. Overview

Our process of automatically learning and extracts both syntactic and semantic features goes through multiple steps (see Figure 4). These features used to build a classifier for vulnerability prediction. We consider each Java source file as consisting of a header (which contains a declaration of class variables) and a set of methods. We treat a header as a special method (method 0). We parse the code within each method into a *sequence of code tokens*, which is fed into into a Long Short-Term Memory (LSTM) system to learn a vector representation of the method (i.e. *method features*). This important step transforms a variable-size sequence of code tokens into a fixed-size feature vector in a multi-dimensional space. In addition, for each input code token, the trained LSTM system also gives us a so-called *token state*, which captures the distributional semantics of the code token in its context of use.

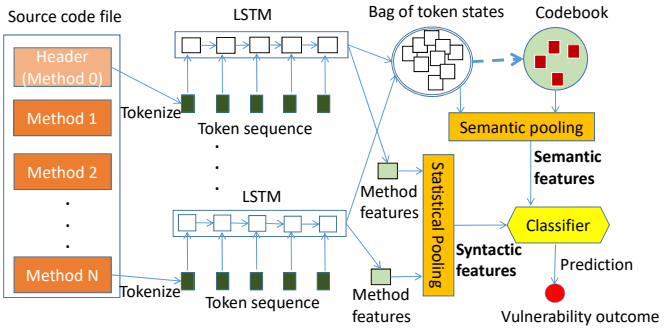

Fig. 4: Overview of our approach for automatic feature learning for vulnerability prediction based on LSTM. The codebook is constructed from all bags of token states in all projects, and the process is detailed in Fig. 6

After this step, we obtain a set of method feature vectors, one for each method in a file. The next step is aggregating those feature vectors into a single feature vector. The aggregation operation is known as *pooling*. For example, the simplest *statistical pooling* method is mean-pooling where we take the sum of the method vectors and divide it by the number of

methods in a file. More complex pooling methods can be used and we will discuss it in more detail. This step produces a set of *syntactic features* for a file.

Those learned syntactic features are however local to a project. For example, method names and variables are typically project-specific. Hence, using only those features alone may be effective for within-project prediction but may not be sufficient for cross-project settings. Our approach therefore learns another set of features to address this generalization issue. To do so, we build up a universal bag of token states from all files across all the studied projects. We then automatically group those code token states into a number of clusters based on their semantic closeness. The centroids in those clusters form a so-called "codebook", which is used for generating a set of *semantic features* for a file through a *semantic pooling* process. We will now describe each of these steps in details.

B. Parsing source code

We use Java Abstract Syntax Tree (AST) to extract syntactic information from source code. To do so, we utilize JavaParser [16] to lexically analyze each source file and obtain an AST. Each source file is parsed into a set of methods and each method is parsed into a sequence of code tokens. All class attributes (i.e. the header) are grouped into a sequence of tokens. Comments and blank lines are ignored. Following standard practice (e.g. as done in [17]), we replace integers, real numbers, exponential notation, and hexadecimal numbers with a generic $\langle num \rangle$ token, and replace constant strings with a generic $\langle str \rangle$ token. We also replace less popular tokens (e.g. occurring only once in the corpus) and tokens which exist in test sets but do not exist in the training set with a special token $\langle unk \rangle$. A fixed-size vocabulary $\mathscr{V}$ is constructed based on top N popular tokens, and rare tokens are assigned to $\langle unk \rangle$. Doing this makes our corpus compact but still provides partial semantic information.

C. Learning code token semantics

After the parsing and tokenizing process, each method is now a sequence of code tokens $\langle w_1, w_2, ..., w_n \rangle$. The files in training set give us a large number sequences of code tokens, which are input to an LSTM system. An LSTM unit takes as input a vector representing a code token. Hence, we need to convert each code token into a fixed-length continuous vector. This process is known as *code token embedding*. We do so by maintaining a token embedding matrix $\mathcal{M} \in \mathbb{R}^{d \times |\mathscr{V}|}$ where d is the size of a code token vector and $|\mathscr{V}|$ is the size of vocabulary $\mathscr{V}$. Each code token has an index in the vocabulary, and this embedding matrix acts as a look-up table: each column i^{th} in the embedding matrix is an embedded vector for the token i^{th}. We denote $\boldsymbol{x}_t$ as a vector representation of code token w_t. For example, token "try" is converted in vector [0.1, 0.3, -0.2] in the example in Figure 5.

The code sequence vectors that make up each method are then input to a sequence of LSTM units. Specifically, each token vector $\boldsymbol{x}_t$ in a sequence $\langle \boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n \rangle$ is input into an LSTM unit (see Figure 5). As LSTM is a recurrent net,

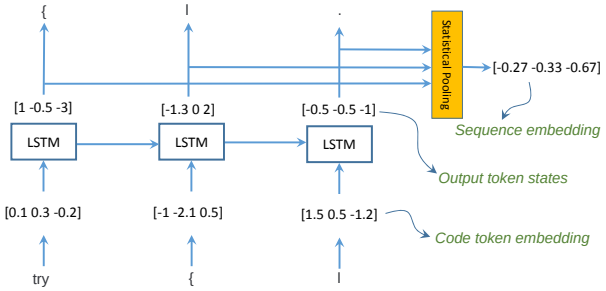

Fig. 5: An example of how a vector representation is obtained for a code sequence

all the LSTM units share the same model parameters. Each unit computes the output state $\boldsymbol{s}_t$ for an input token $\boldsymbol{x}_t$. For example in Figure 5, the output state vector for code token "try" is [0.1, 0.3, -0.2]. The size of this vector can be different from the size of the input token vector (i.e. $d \neq d'$), but for simplicity in training the model we assume they are the same. The state vectors are used to predict the next tokens using another token weight matrix denoted as $\mathcal{U} \in \mathbb{R}^{d' \times |\mathscr{V}|}$.

LSTM automatically learns both model parameters, the token weight matrix $\mathcal{U}$ and the code token embedding matrix $\mathcal{M}$ by maximizing the likelihood of predicting the next code token in the training data. Specifically, we use the output state vector of code token w_t to predict the next code token w_{t+1} from a context of earlier code tokens $w_{1:t}$ by computing a posterior distribution:

$$P\left(w_{t+1} = k \mid w_{1:t}\right) = \frac{\exp\left(\mathcal{U}_k^\top \boldsymbol{s}_t\right)}{\sum\limits_{k'} \exp\left(\mathcal{U}_{k'}^\top \boldsymbol{s}_t\right)} \tag{2}$$

where k is the index of token w_{t+1} in the vocabulary, $\mathcal{U}^\top$ is the transpose of matrix $\mathcal{U}$, and $\mathcal{U}_k^\top$ indicates the vector in column k^{th} of $\mathcal{U}^\top$, and k' runs through all the indices in the vocabulary, i.e. $k' \in \{1, 2, ..., |\mathscr{V}|\}$. This learning style essentially estimates a language model of code. Thus the LSTM automatically learns a grammar of code [18].

1) Model training: LSTM can automatically train itself using code sequences in all the methods extracted from our dataset. During training, for every token in a sequence $\langle w_1, w_2, ..., w_n \rangle$, we know the true next token. For example, the true next token after "try" is "{" in the example Figure 5. We use this information to learn the model parameters which maximize the accuracy of our next token predictions. To measure the accuracy, we use the log-loss (i.e. the cross entropy) of each true next token, i.e. $-\log P(w_1)$ for token w_1, $-\log P(w_2 \mid w_1)$ for token w_2, ..., $-\log P\left(w_n \mid w_{1:n-1}\right)$ for token w_n. The model is then trained using many known sequences of code tokens in a dataset by minimizing the following sum log-loss in each sequence:

$$L(\mathcal{P}) = -\log P(w_1) - \sum_{t=1}^{n-1} \log P\left(w_{t+1} \mid \boldsymbol{w}_{1:t}\right) \tag{3}$$

which is essentially $-\log P\left(w_1, w_2, ..., w_n\right)$.

Learning involves computing the gradient of $L(\mathcal{P})$ during the back propagation phase, and updating the model parameters $\mathcal{P}$, which consists of $\mathcal{M}$, $\mathcal{U}$ and other internal LSTM parameters, via stochastic gradient descent.

2) Generating output token states: Once the training phase has been completed we use the learned LSTM to compute a code token state vector $\boldsymbol{s}_t$ for every code token w_t extracted in our dataset. The use of LSTM ensures that a code token state contains information from other code tokens that come before it. Thus, a code token state captures the *distributional semantics*, a Natural Language Processing concept which dictates that the meaning of a word (code token) is defined by its context of use [19]. The same lexical token can theoretically be realized in infinite number of usage contexts. Hence a token semantics is a point in the semantic space defined by all possible token usages. The token states are then used for generating two distinct sets of features for a file.

D. Generating syntactic features

Generating syntactic features for a file involves two steps. First, we generate a set of features for each method in the file. To do so, we first extract a sequence of code tokens $\langle w_1, w_2, ..., w_n \rangle$ from a method, feed it into the trained LSTM system, and obtain an output sequence of token states $\langle \boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_n \rangle$ (see Section IV-C2). We then compute the method feature vector by aggregating all the token states in the same sequence so that all information from the start to the end of a method is accumulated (see Figure 5). This process is known as *pooling* and there are multiple ways to perform pooling, but the main requirement is that pooling must be length invariant, that is, pooling is not sensitive to variable method lengths. We employ a number of simple but often effective *statistical pooling* methods: (1) Mean pooling, i.e. $\bar{\boldsymbol{s}} = \frac{1}{n}\sum_{t=1}^{n} \boldsymbol{s}_t$; (2) Variance pooling, i.e., $\boldsymbol{\sigma} = \sqrt{\frac{1}{n}\sum_{t=1}^{n} (\boldsymbol{s}_t - \bar{\boldsymbol{s}}) * (\boldsymbol{s}_t - \bar{\boldsymbol{s}})}$, where $*$ denotes element-wise multiplication; and (3) A concatenation of both mean pooling and variance pooling, i.e. $[\bar{\boldsymbol{s}}, \boldsymbol{\sigma}]$.

Since a file contains multiple methods, the next step involves aggregating all these method vectors a single vector for file. We employ again another statistical pooling mechanism to generate a set of syntactic features for the file.

E. Generating semantic features

Syntactic features are useful for within-project vulnerability prediction since they are local to a method and thus tend to be project-specific. To enable effective cross-project prediction, we need another set of features for a file which reflect how the file positions in a semantic space across all projects. We view a file as a set of code token states (generated from the LSTM system), each of which captures the semantic structure of the token usage contexts. This is different from viewing the file as a Bag-of-Words where a code token is nothing but an index in the vocabulary, regardless of its usage. We partition this set of token states into subsets, each of which corresponds to a distinct region in the semantic space. Suppose there are k regions, each file is then represented as a vector of k dimensions. Each dimension is the number of token state vectors that fall into the respective region.

The next question is how to partition the semantic space into a number of regions. To do so, we borrow the concept from computer vision by considering each token in a file as an analogy for a salient point (i.e. the most informative point in an image). The token states are akin to the set of point descriptors such as SIFT [20]. The main difference here is that in vision, visual descriptors are calculated manually, whereas in our setting token states are learnt automatically through LSTM. In vision, descriptors are clustered into a set of points called *codebook* (not to be confused with the software source code), which is essentially the descriptor centroids.

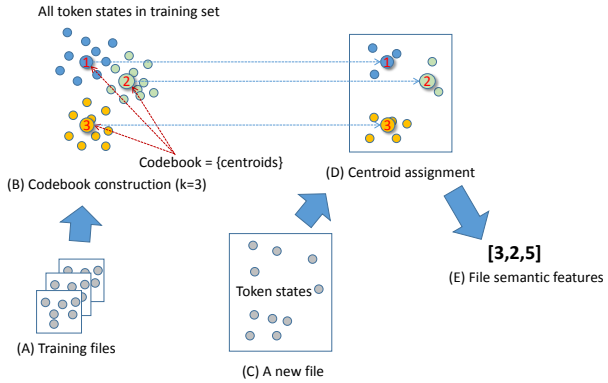

Fig. 6: An example of using "codebook" to automatically learn and generate file semantic features

Similarly, we can build a codebook that summarizes all token states, i.e. the semantic space, across all projects in our dataset. Each "code" in the codebook represents a distinct region in the semantic space. We construct a codebook by using k-means to cluster all state vectors in the training set, where k is the pre-defined number of centroids, and hence the size of the codebook (see Part A in Figure 6, each small circle representing a token state in the space). The example in Figure 6 uses $k = 3$ to produce three state clusters. For each new file, we obtain all the state vectors (Part B in Figure 6) and assign each of them to the closest centroids (Part C in Figure 6). The file is represented as a vector of length k, whose elements are the number of centroid occurrences. For example, the new file in Figure 6 has 10 token vectors. We then compute the distances between those vectors to the three centroids established in the training set. We find that 3 of them are closest to centroid #1, 2 to centroid #2 and 5 to centroid #3. Hence, the feature vector for the new file is [3, 2, 5].

This technique provides a powerful abstraction over a number of tokens in a file. The intuition here is that the number of tokens in an entire dataset could be large but the number of usage context types (i.e. the token state clusters) can be small.

Hence, an file can be characterized by the types of the usage contexts which it contains. This approach offers an efficient and effective way to learn new features for a file from the code tokens constituting it. In effect, a file is a collection of distinct regions in the semantic space of code tokens.

F. Vulnerability prediction

Using the process above, we generate both syntactic and semantic features for each file in both the training data and the test data. We then use the standard process described in Section III-A to build and train vulnerability prediction models. Files in the test data are used to assess the performance of the trained prediction models.

V. Implementation

The proposed approach is implemented in Theano [21] and Keras[22] frameworks, running in Python. Theano supports automatic differentiation of the loss in Eq. (3) and a host of powerful adaptive gradient descent methods. Keras is a wrapper making model building much easier.

A. Training details

In particular we use RMSprop as the optimizer with learning rate of 0.02, and smoothing hyper-parameters: $\rho = 0.99$, $\epsilon = 1e-7$. The model parameters are updated in a stochastic fashion, that is, after every mini-batch of size 50. We use $|\mathscr{V}| = 5,000$ most frequent tokens for learning the code language model discussed in Section . We use *dropout* rate of 0.5 at the hidden output of LSTM layer.

The dataset contains more than 240K sequences, in which, 200k sequences are used for training and the others are used for validation. The LSTM which achieved the best perplexity on validation set was finally kept for feature extraction. The classifier is Random Forest implemented in the scikit-learn toolkit. Hyper-parameters are tuned for best performance include (i) the number of trees, (ii) the maximum depth of a tree, (iii) the minimum number of samples required to split an internal node and (iv) the maximum number of features per tree. The code is run on Intel(R) Xeon(R) CPU E5-2670 0 @ 2.6GHz. There machine has two CPUs, each has 8 physical cores or 16 threads, with a RAM of 128GB.

B. Handling large vocabulary

To evaluate the prediction probability in Eq. (2) we need to iterate through all unique tokens in the vocabulary. Since the vocabulary's size is large, this can be highly expensive. To tackle the issue, we employ an approximate method known as Noise Contrastive Estimation [23], which approximates the vocabulary at each probability evaluation by a small subset of words randomly sampled from the vocabulary. We use 100 words, as it is known to work well in practice [24].

C. Handling long methods

Methods are variable in size. This makes learning inefficient because we need to handle each method separately, not making use of recent advances in Graphical Processing Units (GPUs). A better way is to handle methods in mini-batch of fixed size. A typical way is to pad short methods with dummy tokens so that all methods in the same mini-batch have the same size. However, since some methods are very long, this approach will result in a waste of computational effort to handle dummy tokens. Here we use a simple approach to split a long method into non-overlapping sequences of fixed length T, where $T = 100$ is chosen in this implementation due to the faster learning speed. For simplicity, features of a method are simply the mean features of its sequences.

VI. Evaluation

A. Datasets

To carry out our empirical evaluation, we exploited a publicly available dataset [25] that has been used in previous work [10] for vulnerability prediction. This dataset originally contained 20 popular applications which were collected from F-Droid and Android OS in 2011. The dataset covers a diversity of application domains such as education, book, finance, email, images and games. However, the provided dataset only contained the application names, their versions (and dates), and the file names and their vulnerability labels. It did not have the source code for the files, which is needed for our study. Using the provided file names and version numbers, we then retrieved the relevant source files from the code repository of each application.

TABLE I: Dataset statistics

App	#Versions	#Files	Mean files	Mean LOC	Mean Vuln	% Vuln
Crosswords	16	842	52	12,138	24	0.46
Contacts	6	787	131	39,492	40	0.31
Browser	6	433	72	23,615	27	0.37
Deskclock	6	127	21	4,384	10	0.47
Calendar	6	307	51	21,605	22	0.44
AnkiAndroid	6	275	45	21,234	27	0.59
Mms	6	865	144	35,988	54	0.37
Boardgamegeek	1	46	46	8,800	11	0.24
Gallery2	2	545	272	68,445	75	0.28
Connectbot	2	104	52	14,456	24	0.46
Quicksearchbox	5	605	121	15,580	26	0.22
Coolreader	12	423	35	14,708	17	0.49
Mustard	11	955	86	14,657	41	0.47
K9	19	2,660	140	50,447	65	0.47
Camera	6	457	76	16,337	29	0.38
Fbreader	13	3,450	265	32,545	78	0.30
Email	6	840	140	51,449	75	0.54
Keepassdroid	12	1,580	131	14,827	51	0.39

We could not find the code repository for two applications since they appeared no longer available. For some apps, the number of versions we could retrieve from the code repository is less than that in the original datasets. For example, we retrieve the source files for 16 versions of Crossword while the original dataset had 17 versions. The source files for some older versions were no longer maintained in the code repository. Table I provides some descriptive statistics for 18 apps in our dataset, including the number of versions, the total number of files, the average number of files in a version, the average number of lines of code in a version, the average number of vulnerable files in a version, and the ratio of vulnerable files.

B. Research questions

We followed previous work in vulnerability prediction [10] and aimed to answer the following standard research questions:

1) **RQ1. Within-project prediction**: *Are the automatically learned features using LSTM suitable for building a vulnerability prediction model?*
   To answer this question, we focused on one version (the first one) in each application in our dataset. We ran a cross-fold validation experiments by dividing the files in each application into 10 folds, each of which have the approximately same ratio between vulnerable files and clean files. Each fold is used as the test set and the remaining folds are used for training. As a result, we built 10 different prediction models and the performance indicators are averaged out of the 10 folds.
2) **RQ2. Cross-version prediction**: *How does our proposed approach perform in predicting future releases, i.e. when the model is trained using an older version in application and tested on a newer version in the same application?*
   In this second experiment, all the files in the first version of each application are used to train a prediction model, which is then used to predict vulnerability of the files of all subsequent versions. For example, the first version in the Crosswords app is used to training and each of the remaining 15 versions is used as a test set.
3) **RQ3. Cross-project prediction**: *Is our approach suitable for cross-project predictions where the model is trained using a source application and tested on a different application?*
   In the third experiment, we used all the files in the first version of each project for training a prediction model. This model is then tested on the first version of the remaining applications.

C. Benchmarks

We compare the performance of our approach against the following benchmarks:

Software metrics: Complexity metrics have been extensively used for defect prediction (e.g. [26]) and vulnerability prediction (e.g. [8, 27, 28]). This is resulted from the intuition that complex code is difficult to understand, maintain and test, and thus has a higher chance of having vulnerabilities than simple code. We have implemented a vulnerability prediction models based on 60 metrics. These features are commonly used in existing vulnerability prediction models. They covers 7 categories: cohesion metrics, complexity metrics, coupling metrics, documentation metrics, inheritance metrics, code duplication metrics, and size metrics.

Bag of Words: This technique has been used in previous work [10], which also consider source code as a special form of text. Hence, it treats a source file as a collection of terms associated with frequencies. The term frequencies are the features which are used as the predictors for a vulnerability prediction model. Lexical analysis is done to source code to break it into a vector of code tokens and the frequency of each token in the file is counted. We also followed previous work [10] by discretizing the BoW features since they found that this method significantly improved the performance of the vulnerability prediction models. The discretization process involves transforming the numerical BoW features into two *bins*. If a code token occurs more than a certain threshold (e.g. 5 times) than it is mapped to one bin, otherwise it is mapped to another bin.

Deep Belief Network: Recent work [29] has demonstrated that Deep Belief Network (DBN) [30] worked well for defect prediction. DBN is a family of stochastic deep neural networks which extract multiple layers of data representation. In our implementation, DBN takes the word counts per file as input and produces a latent posterior as output, which is then used as a new file representation. Since the standard DBN accepts only input in the range [0,1], we normalize the word counts by dividing each dimension to its maximum value accross the entire training data. The DBN is then built in a stage-wise fashion as follows. At the bottom layer, a Restricted Boltzmann Machine (RBM) is trained on the normalized word count. An RBM is a special two-layer neural network with binary neurons. Following the standard practice in the literature, the RBM is trained using Contrastive Divergence [30]. After the first RBM is trained, its posterior is used as the input for the next RBM, and the training is repeated. Finally, the two RBMs are stacked on top of each other to form a DBN with two hidden layers. The posterior of the second RBM is used as the new file representation. In our implementation, the two hidden layers have the size of 500 and 128, respectively.

To enable a fair comparison, we used the same classifier for our prediction models and all the benchmarks. We chose Random Forests (RF), an ensemble method which combines the estimates from multiple estimators since it is one of the most effective classifier for vulnerability prediction [10]. We employed the standard Precision, Recall, and F-measure, which has been widely-used in the literature (e.g. [5, 8, 10, 28]) for evaluating the predictive performance of vulnerability prediction models built using the three above benchmarks and our approach.

D. Results

1) Learned code token semantics: An important part of our approach is learning the semantics of code tokens using the context of its usage through LSTM. Figure 7 show the top 2,000 frequent code tokens used in our dataset. They were automatically grouped in 10 clusters (using K-means clustering) based on their token states learned through LSTM. Recall that these clusters are the basis for us to construct a codebook (discussed in Section IV-E). We used t-distributed stochastic neighbor embedding (t-SNE) [31] to display high-dimensional vectors in two dimensions. We show here some representative code tokens from some clusters for a brief illustration. Code tokens that are semantically related are grouped in the same cluster. For example, code tokens related to exceptions such as IllegalArgumentException, FileNotFoundException, and NoSuchMethodException are grouped

TABLE II: Within-project results (RQ1) for the three benchmarks and three variations of our approach. "Joint features" indicates the use of both syntactic features and semantic features.

Application	Software metrics			Bag-of-Words			Deep Belief Network			**Syntactic features**			**Semantic features**			**Joint features**		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
AnkiAndroid	0.62	0.65	0.61	1.00	0.94	0.96	1.00	0.94	0.96	1.00	1.00	1.00	1.00	0.94	0.96	1.00	0.94	0.96
Boardgamegeek	0.18	0.35	0.23	0.95	1.00	0.97	0.95	1.00	0.97	0.95	0.95	0.93	0.90	0.90	0.90	0.90	0.90	0.90
Browser	0.65	0.60	0.62	0.96	1.00	0.98	0.96	1.00	0.98	1.00	1.00	1.00	0.96	1.00	0.98	0.96	1.00	0.98
Calendar	0.52	0.43	0.45	0.89	0.80	0.83	1.00	0.91	0.94	1.00	1.00	1.00	0.96	0.94	0.94	1.00	1.00	1.00
Camera	0.72	0.81	0.73	0.82	0.92	0.86	0.83	0.93	0.87	0.88	1.00	0.93	0.92	0.98	0.94	0.92	0.98	0.94
Connectbot	0.65	0.73	0.64	0.97	0.90	0.91	0.92	0.90	0.88	1.00	0.97	0.98	0.93	0.93	0.93	0.97	0.93	0.93
Contacts	0.72	0.72	0.71	0.94	0.88	0.89	0.92	0.81	0.83	0.96	0.94	0.95	0.89	0.94	0.89	0.87	0.97	0.90
Coolreader	0.55	0.70	0.60	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.94	0.96	1.00	0.94	0.96	1.00	0.94	0.96
Crosswords	0.55	0.57	0.55	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.83	0.85	0.89	0.83	0.85
Deskclock	0.30	0.20	0.23	0.92	0.92	0.89	0.83	0.92	0.83	1.00	1.00	1.00	1.00	0.92	0.94	1.00	0.92	0.94
Email	0.83	0.84	0.82	0.92	0.93	0.92	0.96	0.94	0.94	0.93	0.95	0.93	0.95	0.90	0.92	0.94	0.93	0.93
Fbreader	0.60	0.52	0.55	0.78	0.88	0.81	0.81	0.85	0.82	0.83	0.82	0.82	0.85	0.83	0.84	0.82	0.86	0.83
Gallery2	0.62	0.60	0.57	0.80	0.91	0.84	0.75	0.87	0.78	0.83	0.88	0.84	0.83	0.87	0.84	0.80	0.89	0.83
K9	0.73	0.82	0.76	0.94	0.88	0.89	0.96	0.87	0.90	0.89	1.00	0.94	0.88	0.97	0.91	0.88	0.98	0.92
Keepassdroid	0.69	0.75	0.71	0.91	0.88	0.88	0.85	0.90	0.87	0.93	0.97	0.94	0.90	0.91	0.89	0.88	0.89	0.88
Mms	0.78	0.60	0.67	0.86	0.85	0.85	0.84	0.85	0.84	0.87	0.85	0.85	0.87	0.88	0.87	0.87	0.92	0.89
Mustard	0.81	0.68	0.72	0.94	0.93	0.93	0.94	0.87	0.89	0.96	0.99	0.97	0.97	0.93	0.94	0.95	0.99	0.96
Quicksearchbox	0.60	0.42	0.43	0.84	0.90	0.83	0.83	0.90	0.84	0.93	0.90	0.88	0.89	0.77	0.82	0.89	0.79	0.82
Average	0.62	0.61	0.59	0.91	0.91	0.90	0.90	0.91	0.89	**0.94**	**0.95**	**0.93**	**0.92**	**0.91**	**0.91**	**0.92**	**0.93**	**0.91**

in one cluster. This indicates, to some extent, that the learned token states effectively capture the semantic relations between code tokens, which is useful for us to learn both syntactic and semantic features later.

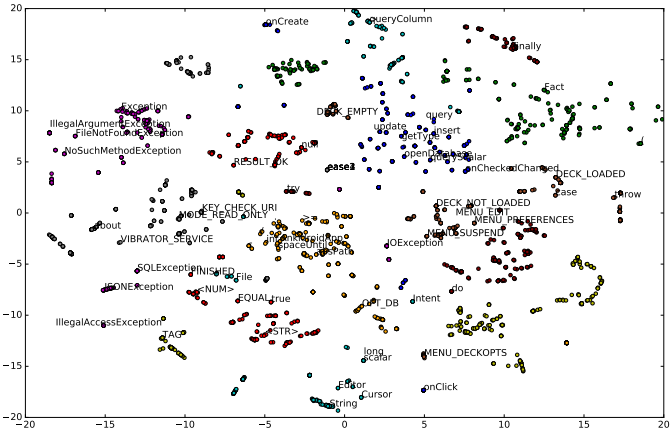

Fig. 7: Top 2,000 frequent code tokens were automatically grouped into clusters (each cluster has a distinct color)

2) Within-project prediction (RQ1): We experimented with a number of variations of our approach by varying the pooling techniques (mean pooling and standard deviation pooling) and the use of syntactic and semantic features. Table II reports the precision, recall and F-measure for the three benchmarks (software metrics, Bag-of-Words, and Deep Belief Network) and three variations of our approach: using only syntactic features, using only semantic features, and using a joint set of both features types. Note that both the syntactic feature option and and the joint feature option we reported here used mean pooling for generating method features and standard deviation pooling for generating syntactic file features. The full report which contains the results for all variations of pooling is available at https://goo.gl/OnRnPp.

Our approach obtained a strong result: in *all* 18 applications, they all achieved more than 80% in precision, recall, and F-measure (see Table II). Among the three variation of our approach, using only syntactic features appeared to be the best option for vulnerable prediction within one version of a project. This option delivered the highest precision, recall and F-measure averaging across 18 applications. This result is consistent with our underlying theory that syntactic features are project-specific and are thus useful for within-project prediction. Our approach outperforms the three benchmarks to varying extents. The syntactic feature option offered 58% F-measure improvement over the software metrics approach. Both Bag-of-Words (BoW) and Deep Belief Network approaches also achieved very good results (on average 89–90% F-measure) – which is consistent with the findings in previous work [10] in case of BoW. Hence, comparing against these two benchmarks, the improvement brought by our approach is small (approximately 3% in F-measure).

3) Cross-version prediction (RQ2): Table III reports the results in a cross-version setting where a prediction model was trained using the first version of an application and tested using the subsequent versions of the same applications. Our approach again slightly outperforms the BoW and DBN benchmarks. Among the three variations, using semantic features (alone or joint with syntactic features) becomes the best options. This is because semantic measures improve the generalization of the prediction models, which becomes slightly more useful in cross-version settings.

4) Cross-project prediction (RQ3): This experiment followed the setup in previous work [10]. We first built 18 prediction models, each of which use the first version of each application for training. Each model was then tested using the first version of the other 17 applications. We ran this experiment with the three benchmarks and variations of our approach. We used the same baseline as in previous work

TABLE III: Cross-version results (RQ2) for the three benchmarks and three variations of our approach. "Joint features" indicates the use of both syntactic features and semantic features.

Application	Software metrics			Bag-of-Words			Deep Belief Network			**Syntactic features**			**Semantic features**			**Joint features**		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
AnkiAndroid	0.80	0.71	0.75	0.88	0.85	0.87	0.87	0.85	0.86	0.83	0.90	0.87	0.91	0.84	0.87	0.89	0.87	0.88
Browser	0.69	0.65	0.66	0.75	0.63	0.68	0.75	0.65	0.70	0.59	0.64	0.61	0.84	0.57	0.68	0.82	0.58	0.68
Calendar	0.69	0.77	0.72	0.74	0.83	0.78	0.71	0.85	0.78	0.73	0.81	0.77	0.79	0.89	0.83	0.79	0.84	0.82
Camera	0.51	0.81	0.62	0.60	0.90	0.72	0.58	0.89	0.70	0.65	0.84	0.73	0.67	0.88	0.76	0.84	0.80	0.82
Connectbot	0.75	0.75	0.75	1.00	0.96	0.98	1.00	0.96	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Contacts	0.68	0.72	0.70	0.74	0.71	0.73	0.73	0.75	0.74	0.60	0.80	0.69	0.69	0.72	0.70	0.73	0.69	0.71
Coolreader	0.80	0.77	0.78	0.93	0.79	0.85	0.83	0.82	0.83	0.94	0.89	0.91	0.89	0.85	0.87	0.92	0.86	0.89
Crosswords	0.87	0.72	0.79	0.90	0.81	0.86	0.83	0.88	0.85	0.85	0.89	0.87	0.88	0.90	0.89	0.92	0.83	0.87
Deskclock	0.87	0.66	0.75	0.89	0.80	0.85	0.87	0.76	0.81	0.93	0.80	0.86	0.90	0.86	0.88	0.89	0.82	0.86
Email	0.74	0.83	0.78	0.82	0.83	0.82	0.79	0.85	0.82	0.77	0.93	0.84	0.86	0.89	0.87	0.83	0.90	0.86
Fbreader	0.83	0.75	0.78	0.79	0.83	0.81	0.80	0.75	0.77	0.84	0.81	0.83	0.91	0.84	0.88	0.86	0.84	0.85
Gallery2	0.74	0.74	0.74	0.96	0.93	0.95	0.88	0.88	0.88	0.95	0.97	0.96	0.96	0.99	0.97	0.96	0.97	0.97
K9	0.77	0.89	0.82	0.85	0.84	0.84	0.84	0.86	0.85	0.80	0.94	0.87	0.84	0.98	0.90	0.89	0.92	0.90
Keepassdroid	0.92	0.91	0.91	0.92	0.87	0.89	0.86	0.86	0.86	1.00	0.99	0.99	0.98	0.91	0.94	0.99	0.92	0.95
Mms	0.83	0.81	0.82	0.93	0.90	0.91	0.94	0.86	0.90	0.93	0.89	0.91	0.93	0.93	0.93	0.92	0.96	0.94
Mustard	0.93	0.90	0.91	0.97	0.96	0.96	0.92	0.93	0.93	0.98	0.98	0.98	0.99	0.95	0.97	0.99	0.95	0.97
Quicksearchbox	0.57	0.69	0.62	0.71	0.73	0.72	0.56	0.79	0.66	0.76	0.80	0.78	0.74	0.87	0.80	0.77	0.88	0.82
Average	0.76	0.77	0.76	0.85	0.83	0.84	0.81	0.84	0.82	0.83	**0.87**	**0.85**	**0.87**	**0.87**	**0.87**	**0.88**	**0.86**	**0.87**

[10]: a model is applicable to a tested application if both precision and recall are above 80%. For each application, Table IV reports the number of other applications to which the corresponding models can be applied.

TABLE IV: Cross-project results (RQ3) for the three benchmarks and three variations of our approach.

App	Metrics	BoW	DBN	**Syntactic**	**Semantic**	**Joint**
AnkiAndroid	1	2	2	3	7	5
Boardgamegeek	0	1	1	0	1	1
Browser	0	1	2	0	0	1
Calendar	0	1	1	1	4	5
Camera	1	3	2	3	6	6
Connectbot	0	2	2	4	4	5
Contacts	0	1	3	1	2	2
Coolreader	1	2	3	4	4	6
Crosswords	0	3	3	2	1	1
Deskclock	0	1	1	0	1	1
Email	1	2	2	4	4	4
Fbreader	0	2	2	4	6	6
Gallery2	0	3	2	3	3	2
K9	1	3	3	2	7	8
Keepassdroid	0	3	1	4	2	4
Mms	0	4	2	3	5	6
Mustard	0	3	3	3	8	8
Quicksearchbox	0	2	2	1	3	3
Average	0.3	2.2	2.1	**2.3**	**3.8**	**4.1**

The results suggest that using semantic features really improve the general applicability of prediction models. All the models using both semantic and syntactic features were successfully applicable to at least one other application. Some of them (e.g. K9 and Mustard) are even applicable to 8 other applications. In this cross-project prediction setting, our approach also offers bigger improvements over the BoW and DBN benchmarks. On average, a joint-feature model is applicable to around 4 other applications, which is 85% improvement compared with BoW or DBN models.

E. Remarks and implications

The high performance of BoW on within-project prediction (RQ1 and RQ2) is not totally surprising for two reasons. One is that BoW has been known as a strong representation for text classification, and source code is also a type of text written in programming languages. The other reason is that although the train files and test files are not identical, a project typically has many versions, and the new versions of the same file may carry a significant amount of information from the old versions. The repeated information can come from multiple forms: fully repeated pieces of code, the same BoW statsistics, or the same code convention and style. Thus any representation that is sufficiently expressive and coupled with highly flexible classifiers such as Random Forests, will likely to work well.

However, this is not the case for cross-project prediction (RQ3). This is when the BoW statistics are likely to be different between projects, and knowledge learned from one project may not transfer well to others. In machine learning and data mining, this problem is known as *domain adaptation*, where each project is a domain. The common approach is to learn the common representation across domains, upon which classifiers will be built. This is precisely what is done using the LSTM-based language model and codebook construction. Note that the LSTM and codebook are learned using all available data without supervision. This suggests that we can actually use external data, even if there are no vulnerability labels. The competitive performance of the proposed deep learning approach clearly demonstrates the effectiveness of this representation learning.

To conclude, when doing within-project prediction, it is useful to use BoW due to its simplicity. But when generalizing from one project to another, it is better to use representation learning. We recommend using LSTM for language model, and codebook for semantic structure discovery.

VII. Threats to validity

There are a number of threats to the validity of our study, which we discuss below.

Construct validity: We mitigated the construct validity concerns by using a publicly available dataset that has been used in previous work [10]. The dataset contains real Android applications and vulnerability labels of the files in those applications. The original dataset did not unfortunately contain the source files. However, we have carefully used the information (e.g. application details, version numbers and date) provided with the dataset to retrieve the relevant source files from the code repository of those applications.

Conclusion validity: We tried to minimize threats to conclusion validity by using standard performance measures for vulnerability prediction [8, 10, 28, 32]. We however acknowledge that a number of statistical tests [33] can be applied to verify the statistical significance of our conclusions. Although we have not seen those statistical tests being used in previous work in vulnerability prediction, we plan to do this investigation in our future work.

Internal validity: The dataset we used contains vulnerability labels only for Java source files. In practice, other files (e.g. XML manifest files) may contain security information such as access rights. Another threat concerns the cross-version prediction where we replicated the experiment done in [10] and allowed that the exactly same files might be present between versions. This might have inflated the results, but all the prediction models which we compared against in our experiment benefit from this.

External validity: We have considered a large number of applications which differ significantly in size, complexity, domain, popularity and revision history. We however acknowledge that our data set may not be representative of all kinds of Android applications. Further investigation to confirm our findings for other Android applications as well as other types of applications such as web applications and applications written in other programming languages such as PhP and C++.

VIII. Related work

Machine learning techniques have been widely used to build vulnerability prediction models. Early approaches (e.g. [27]) employed complexity metrics such as (e.g. McCabe's cyclomatic complexity, nesting complexity, and size) as the predictors. Later approaches enriched this software metric feature set with coupling and cohesion metrics (e.g. [28]), code churn and developer activity metrics (e.g. [8])), and dependencies and organizational measures (e.g. [5]). Those approaches require knowledgeable domain experts to determine the metrics that are used as predictors for vulnerability.

Recent approaches treat source code as another form of text and leverage text mining techniques to extract the features for building vulnerability prediction models. The work in [10] used the Bag-of-Words representation in which a source code file is viewed as a set of terms with associated frequencies. They then used the term-frequencies as the features for predicting vulnerability. The BoW approach eliminates the need for manually designing the features. BoW models also produced higher recall than software metric models for PHP applications [32]. However, the BoW approaches carries the inherent limitation of BoW in which syntactic information such as code order is disregarded.

Predicting vulnerabilities is related to software defect prediction. The study in [34] found that some defect prediction models can be adapted for vulnerability prediction. While code metrics were commonly used as features for building defect prediction models [26], various other metrics have also been employed such as change-related metrics [35, 36], developer-related metrics [37], organization metrics [38], and change process metrics [39]. Recently, a number of approaches (e.g. [29, 40]) have leveraged a deep learning model called Deep Belief Network [41] to automatically learn features for defect prediction.

Deep learning has recently attracted increasing interests in software engineering. The work in [42] proposes generic deep learning framework based on LSTM for modeling software and its development process. The work in [17] demonstrated the effectiveness of using recurrent neural networks (RNN) to model source code. Their later work [43] extended these RNN models for detecting code clones. The work in [44] uses a special RNN Encoder–Decoder, which consists of an encoder RNN to process the input sequence and a decoder RNN with attention to generate the output sequence, to generate API usage sequences for a given API-related natural language query. The work in [45] also uses RNN Encoder–Decoder but for fixing common errors in C programs. The work [24] showed that LSTM is even more effective in code modeling, which inspired us to use it for learning vulnerability features. The work in [46] uses Convolutional Neural Networks (CNN) [47] for bug localization.

IX. Conclusions and future work

This paper proposes to leverage Long-Short Term Memory, a representation deep learning model, to automatically learn features directly from source code for vulnerability prediction. The learned syntactic features capture the sequential structure in code at the method level, while semantic features characterize a source code file by usage contexts of its code tokens. We performed an evaluation on 18 Android applications from a public dataset provided in previous work [10]. The results for within-project prediction demonstrate that the automatically learned features significantly outperforms the traditional software metrics approach (58% improvement on average), and offers a small improvement (3% on average) over the Bag-of-Word approach and another deep learning approach (Deep Belief Network). For cross-project prediction, the results suggest that our approach is clearly superior to these state-of-the-art techniques (85% improvement on average).

Our future work involves applying the proposed approach to other types of applications (e.g. Web applications) and programming languages (e.g. PHP or C++) where vulnerability datasets are available. We also aim to leverage our approach to learn features for predicting vulnerabilities at the method and code change levels. In addition, we plan to explore how our approach can be extended to predicting general defects and safety-critical hazards in code. Finally, our future investigation

involves building a fully end-to-end prediction system from raw input data (code tokens) to vulnerability outcomes.

ACKNOWLEDGEMENT

The paper is supported by Samsung 2016 Global Research Outreach Program (GRO) entitled "Predicting hazardous software components using deep learning".

REFERENCES

[1] R. Hackett, "On Heartbleed's anniversary, 3 of 4 big companies are still vulnerable," Fortunue, http://fortune.com/2015/04/07/heartbleed-anniversary-vulnerable, April 2015.

[2] McAfee, C. for Strategic, and I. Studies, "Net Losses: Estimating the Global Cost of Cybercrime," June 2014.

[3] C. Ventures, "Cybersecurity market report," http://cybersecurityventures.com/cybersecurity-market-report, Accessed on 01 May 2017, March 2017.

[4] C. Williams, "Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug," The Register, http://www.theregister.co.uk/2014/04/09/heartbleed_explained, Accessed on 01 May 2017, April 2014.

[5] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 421–428. [Online]. Available: http://dx.doi.org/10.1109/ICST.2010.32

[6] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 97–106. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2011.18

[7] M. Ceccato and R. Scandariato, "Static analysis and penetration testing from the perspective of maintenance teams," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. New York, NY, USA: ACM, 2016, pp. 25:1–25:6. [Online]. Available: http://doi.acm.org/10.1145/2961111.2962611

[8] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.81

[9] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[10] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining." *IEEE Trans. Software Eng.*, vol. 40, no. 10, pp. 993–1006, 2014. [Online]. Available: http://dblp.uni-trier.de/db/journals/tse/tse40.html#ScandariatoWHJ14

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] D. Mohindra, "SEI CERT Oracle Coding Standard for Java," https://www.securecoding.cert.org/confluence/display/java/LCK08-J.+Ensure+actively+held+locks+are+released+on+exceptional+conditions, Accessed on 01 May 2017.

[14] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[15] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[16] JavaParser, "JavaParser," http://javaparser.org, Accessed on 01 Feb 2017.

[17] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345.

[18] C. L. Giles, S. Lawrence, and A. C. Tsoi, "Noisy time series prediction using recurrent neural networks and grammatical inference," *Machine learning*, vol. 44, no. 1, pp. 161–183, 2001.

[19] M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors." in *ACL (1)*, 2014, pp. 238–247.

[20] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2. Ieee, 1999, pp. 1150–1157.

[21] Theano, "Theano," http://deeplearning.net/software/theano/, Accessed on 01 May 2017.

[22] Keras, "Keras: Deep Learning library for Theano and TensorFlow," https://keras.io/, Accessed on 01 May 2017.

[23] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 307–361, 2012.

[24] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," in *Workshop on Naturalness of Software (NL+SE), co-located with the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[25] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Android study dataset," https://sites.google.com/site/textminingandroid, Accessed on 15 Jan 2017, 2014.

[26] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.103

[27] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 315–317. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414065

[28] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2010.06.003

[29] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[30] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[31] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 2579-2605, p. 85, 2008.

[32] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ser. ISSRE '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 23–33. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2014.32

[33] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[34] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9190-8

[35] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[36] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th*

International Conference on Software Engineering, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[37] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453105

[38] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368160

[39] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[40] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, ser. QRS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 17–26. [Online]. Available: http://dx.doi.org/10.1109/QRS.2015.14

[41] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504 – 507, 2006.

[42] H. K. Dam, T. Tran, J. Grundy, and A. Ghose, "DeepSoft: A vision for a deep model of software," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16. ACM, To Appear., 2016.

[43] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[44] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[45] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[46] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 1606–1612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060832.3060845

[47] Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Handwritten Digit Recognition with a Back-propagation Network, pp. 396–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.109279