Automatic Feature Learning for Predicting Vulnerable Software Components

Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose

Abstract—Code flaws or vulnerabilities are prevalent in software systems and can potentially cause a variety of problems including deadlock, hacking, information loss and 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 code features (e.g., complexity metrics or frequencies of code tokens) that represent the characteristics of the potentially problematic code to locate. 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 of code. Our evaluation on 18 Android applications and the Firefox application demonstrates that the prediction power obtained from our learned features is better than what is achieved by state of the art vulnerability prediction models, for both within-project prediction and cross-project prediction.

Index Terms—Software vulnerability prediction, mining software engineering repositories, empirical software engineering

1 Introduction

A software vulnerability – a security flaw, glitch, bug, or weakness found in software systems – can potentially cause significant damage 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 very difficult as 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 the remaining parts of the software code which are 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 [8].

A common approach to building vulnerability prediction models is by using machine learning techniques. A number of features representing software code are selected for use as predictors for vulnerability [5], [9], [10], [11], [12], [13], [14]. The most commonly used features in previous work are software metrics (e.g., size of code, number of dependencies, and cyclomatic complexity) (e.g., [11]), code churn metrics (e.g., the number of code lines changed) and developer activity (e.g., [10]), and dependencies and organizational measures (e.g., [5]). However, using those features does not help us recognize the code that is semantically different and hence potentially vulnerable [15]. In many cases, two pieces of code may have the same complexity metrics but they behave differently and thus have a different likelihood of vulnerability

- *H. K. Dam, S. W. Ng, and A. Ghose are with the School of Computing and Information Technology, Faculty of Engineering and Information Sciences, University of Wollongong, Wollongong, NSW 2522, Australia. E-mail: {hoa, swn881, aditya}@uow.edu.au.*
- *T. Tran and T. Pham are with the School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia. E-mail: {truyen.tran, phtra}@deakin.edu.au.*
- *J. Grundy is with the Faculty of Information Technology, Monash University, Clayton, Victoria 3800, Australia. E-mail: john.grundy@monash.edu.au.*

Manuscript received 9 Oct. 2017; revised 28 Oct. 2018; accepted 6 Nov. 2018. Date of publication 19 Nov. 2018; date of current version 8 Jan. 2021. (Corresponding author: Hoa Khanh Dam.) Recommended for acceptance by M. Di Penta. Digital Object Identifier no. 10.1109/TSE.2018.2881961

to attack. Furthermore, the choice of which features are selected as predictors is *manually* chosen by knowledgeable domain experts. This has the disadvantage that it may carry outdated experience and underlying biases. In addition, in many situations the handcrafted features do not generalize well: features that work well for a certain software project may not perform as well in other projects [16].

An emerging approach is treating software code as a form of text and leveraging Natural Language Processing (NLP) techniques to automatically extract features. Recent work (e.g., [8]) 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. BoW features rely on the code tokens used by the developers, and thus they are not fixed or pre-determined (as seen in the software metric model). However, the BoW approach has two major weaknesses. First, it ignores the semantics of code tokens, e.g., fails to recognize the semantic relations between "for" and "while". Second, a bag of code tokens does not necessarily capture the semantic structure of code, especially its sequential nature.

Software programs not only follow a well-defined syntax, but also have semantics which describe what the programs mean and how they execute. Thus, approaches that do not capture the semantics of code structure or individual code tokens may miss important information about the programs [17]. In fact, previous studies have demonstrated that semantic information hidden in a program is useful for various software engineering tasks such as code completion, bug detection and defect prediction [15], [18], [19], [20], [21]. This semantic information can also help provide richer representations for vulnerable code and thus improve vulnerability prediction.

The recent advances of deep learning techniques [22] 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) [23], a special kind of recurrent neural network 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 [22] .

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 that do not immediately follow each other in the textual code files. Our previous work [24] has provided a preliminary demonstration of the effectiveness of a language model based on LSTM. However, that work merely sketched the use of LSTM in predicting the next code tokens. Our current work in this paper presents a comprehensive framework where features are learned and combined in a novel way, and are then used in a novel application, i.e., building vulnerability prediction models. The learned features 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 [8] for the Android OS platform and Firefox application [12] from public datasets demonstrate that our approach is highly effective in predicting vulnerabilities in code.

```
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.

The outline of this paper is as follows. In the next section, we provide a motivation example. Section 3 provides a brief background on vulnerability prediction and the neural networks used in our model. We then present an overview of our approach in Section 4, the details of how features are automatically learned in Section 5, and the implementation our approach in Section 6. We report the experiments to evaluate it in Section 7, and discuss the threats to validity in Section 8. In Section 9, we discuss related work before summarizing the contributions of the paper and outlines future work in Section 10.

2 Motivation

In this section, we present a motivating example which demonstrates some major limitations of existing approaches in representing software components for vulnerability prediction. Fig. 1 shows two code listings in Java that we adapted from [25]. 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 in this example). They do so by using a reentrant mutual exclusion lock l in order to enforce exclusive access to the file f. Here, a thread executing this code means to: (i) acquire the lock before reading file f; and then (ii) release the lock when it finishes reading the file so that other threads are able to access the file.

The use of such locking can however result in deadlocks. Listing 1 in Fig. 1 demonstrates an example of deadlock vulnerability. 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 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 Fig. 1 rectifies this vulnerability. It fixes the problem of the lock not being released by calling $unlock()$ in

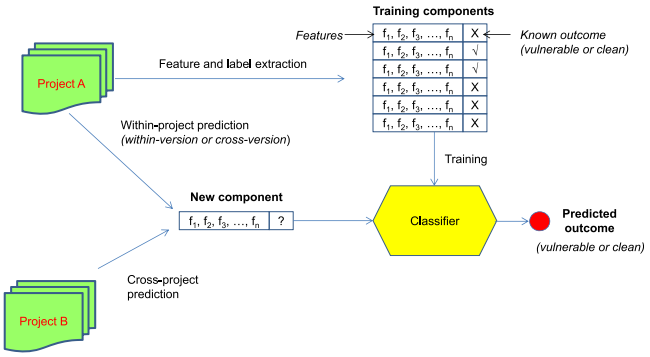

Fig. 2. Vulnerability prediction (adapted from [15]).

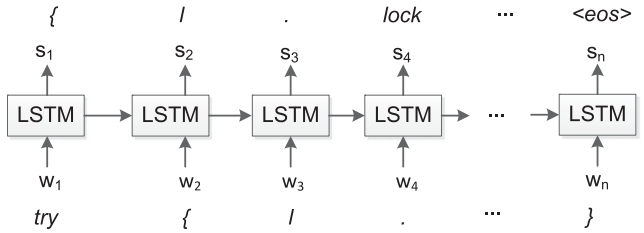

Fig. 3. A recurrent neural network.

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 to code-level vulnerability detection. 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 for a vulnerability detection or analysis recommendation approach. 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.

3 Background

3.1 Vulnerability Prediction

Vulnerability prediction involves determining whether a software component is likely to be vulnerable or not. Most of existing work (e.g., [5], [8], [9], [10], [11], [26], [27], [28]) in vulnerability prediction refer to a component as a source file (e.g., a ".java" file) in a software system. Hence, this level of granularity has become a standard in the literature of predicting vulnerabilities in software code in terms of both benchmark techniques and datasets. The objective here is to alert software engineers with parts of the software system require special focus (e.g., manual inspection or running targeted test case suites), rather than pinpointing exactly the code line(s) where a vulnerability resides [8]. Hence, we also chose to work at the level of files since this is also the scope of most existing work (e.g., [8], [10]) with which we would like to compare our approach against.

Determining if a component is likely to be 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 component is likely to be vulnerable, while *false* indicates that the component is likely to be clean. Vulnerability prediction is therefore to approximate this classification function $vuln(x)$ by learning from a number of examples (i.e., components known to be vulnerable or clean) provided in a *training set* (see Fig. 2). After training, the learned function (or also referred to as the model) is used to automatically determine the vulnerability of new components in the same project (within-project prediction) or in a different project (cross-project prediction). Within-project prediction also has two settings: within-version (new components are from the same version as the training components) and cross-version (new components are from a later version).

To date, various 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).

3.2 Long Short Term Memory

The feature vector representation of file x is critical in building an accurate vulnerability prediction model. While high-level representations such as code complexity metrics are useful, they do not reveal the semantics hidden deeply in source code (as demonstrated in the motivation example in Section 2). Long Short-Term Memory, a deep learning architecture, offers a powerful representation of source code. It is able to automatically learn both syntactic and semantic features which represent long-term dependencies (e.g., a code element may depend on other code elements which are not immediately before it) in source code.

Long Short-Term Memory (LSTM) [23], [29] is a recurrent neural network [30], which maps a sequence of input vectors into a sequence of output vectors (see Fig. 3). A Long Short-Term Memory (LSTM) neural network architecture is a special variant of a Recurrent Neural Network (RNN), which is capable of learning long-term dependencies. This is the key difference from a feedforward neural network which maps an input vector into an output vector.

An LSTM network can be considered as a sequence of LSTM units. Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ be the input sequence (e.g., code tokens), which has a sequence of corresponding labels (e.g., the next code tokens). At each step t, an LSTM unit reads the input $\boldsymbol{w}_t$, the previous output state $\boldsymbol{s}_{t-1}$ and the previous memory c_{t-1} and uses a set of model parameters to compute the output state $\boldsymbol{s}_t$. The output state is used to predict the output (e.g., the next code token based on the previous ones) at each step t.

Each LSTM unit has a *memory cell* c_t which stores accumulated memory of the context (see Fig. 4). This is the key feature allowing an LSTM model to learn long-term dependencies. The information stored in the memory is refreshed at each time step through partially forgetting old, irrelevant information and accepting fresh new input. Specifically, 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

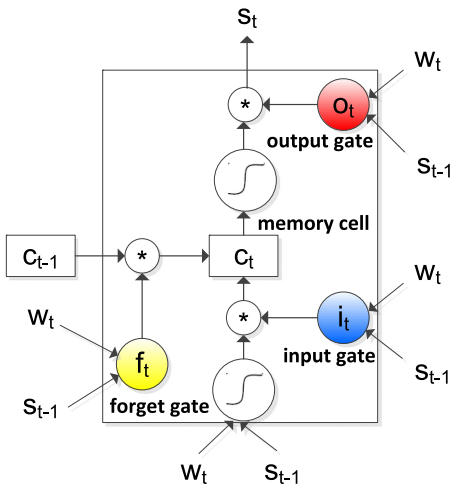

Fig. 4. The internal structure of an LSTM unit.

returns a value between 0 (i.e., complete blockage) and 1 (full passing through).

All those gates are *learnable*, i.e., are trained with the whole code corpus. All LSTM units in the same network share the same parameters since they perform the same task (e.g., predicting the next code token), just with different inputs. Hence, comparing this to traditional feedforward networks, using an LSTM network significantly reduces the total number of model parameters which we need to learn.

An LSTM 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 (i.e., backpropagation) during training to adjust the model parameters such that the errors are minimized. More details about LSTMs can be found in the seminal paper [23].

LSTM is highly effective in learning representations of sequential data, such as natural text and speech, as demonstrated in many recent breakthroughs in machine translation and speed recognition [22]. Since software code is also typically produced by humans, it shares many important properties with natural language text (e.g., repetitive, predictable, and long-term dependencies) [20]. In this study we investigate how a LSTM can be used to learn representations of software code and then use these for vulnerability prediction.

4 Architectural Overview

Our process of automatic feature learning goes through multiple steps (see Fig. 5). 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* (step 1 in Fig. 5), which is fed into a Long Short-Term Memory (LSTM) system to learn a vector representation of the method (i.e., *method features* – step 2 in Fig. 5). 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.

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 (step 3 in Fig. 5). The aggregation operation is known as *pooling*. Pooling aims to transform the joint feature representation (e.g., method features) into a new, more usable one (e.g., file features) that maintain important information while removing irrelevant detail. 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 these in more detail. This step produces a set of *local features* for a file.

Those learned features are however local to a project. For example, method names and variables are typically

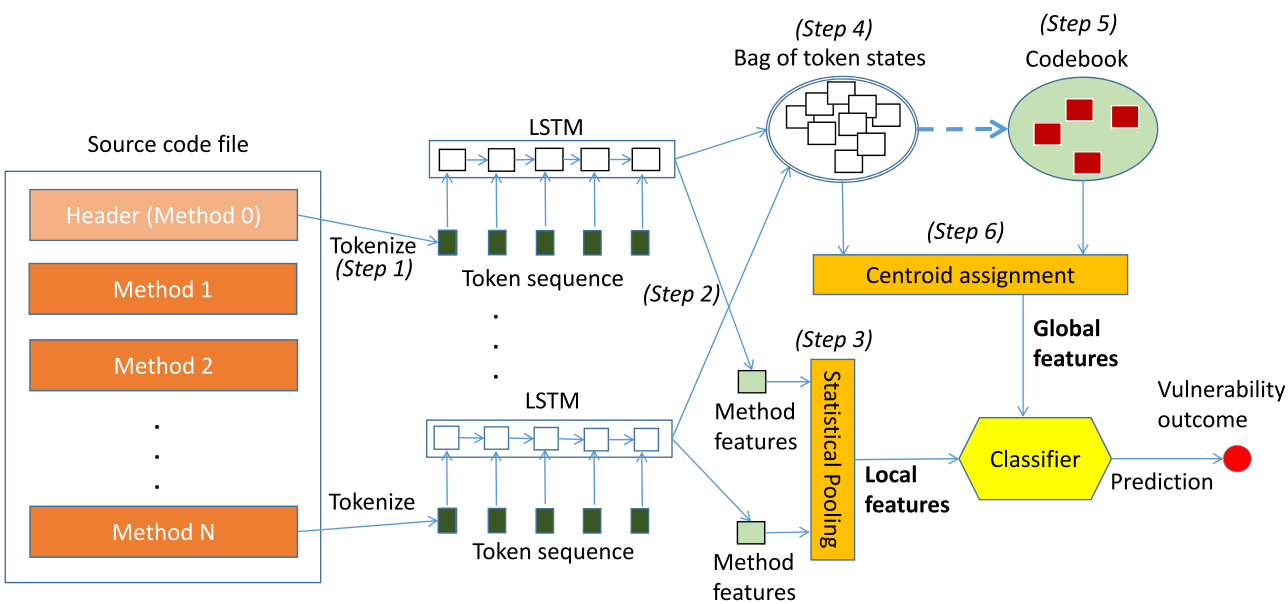

Fig. 5. Overview of our approach for automatic feature learning for vulnerability prediction based on LSTM. The codebook is constructed from all bags of token states in all projects, and the process is detailed in Fig. 7.

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 (step 4 in Fig. 5). We then automatically group those code token states into a number of clusters based on their semantic closeness (step 5). The centroids in those clusters form a so-called "codebook", which is used for generating a set of *global features* for a file through a *centroid assignment* process (step 6). The two sets of learned features are fed into a classifier, which is then trained to predict vulnerable components.

Vulnerability prediction in new projects is often very difficult due to the lack of suitable training data. One common technique to address this issue is training a model using data from a (source) project and applying it to the new (target) project. Since our approach requires only the source code of the source and target projects, it is readily applicable to both within-project prediction as well as for cross-project prediction.

5 Feature Learning, Generation, and Usage

In this section, we describe in detail how our approach automatically learns and generates features representing a source code file and uses them for vulnerability prediction and recommendation to software engineers.

5.1 Parsing Source Code

To use our approach we must convert programs into vectors for our LSTM. To begin, we build Abstract Syntax Trees (AST) to extract key syntactic information from the source code of each source file in a project. To do so, we utilize a parser 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.

During this processing comments and blank lines are ignored as they do not contribute to the actual behaviour of the code. 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. Doing this allows us to generalize from the numbers and strings that are specific to a source file or project. 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 $\langle unk \rangle$ since learning is limited for these tokens. 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 to the computation costs, but still provides sufficient semantic information.

5.2 Learning Code Token Semantics

After the parsing and tokenizing process, each method is now a sequence of code tokens $\langle w_1, w_2, \ldots, w_n \rangle$. We then perform the following steps to learn a vector representation for each code token.

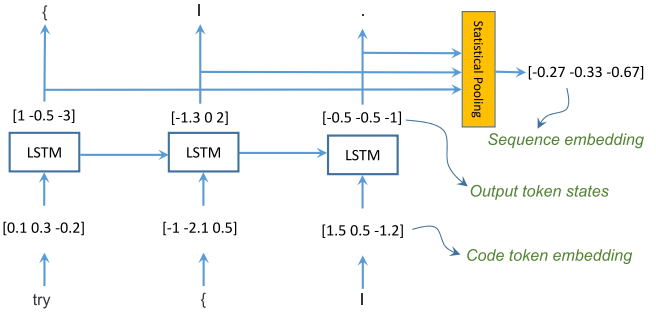

Fig. 6. An example of how a vector representation is obtained for a code sequence.

1) We first represent each code token as a low dimensional, continuous and real-valued vector. This process is known as code token embedding, which is described in detail in Section 5.2.1.
2) Each sequence of code tokens is then input into a sequence of LSTM units. We then train each LSTM unit by predicting the next code token in a sequence (see Section 5.2.2 for details).
3) The trained LSTM is used to generate an output vector (the so-called code token state) for each code token. This vector representation captures the distribution of semantics of a code token in terms of its context of use (see Section 5.2.3 for details).

5.2.1 Code Token Embedding

An LSTM unit takes as its 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 ith in the embedding matrix is an embedded vector for the token ith. 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 Fig. 6.

5.2.2 Model Training

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 $\langle \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n \rangle$ is input into an LSTM unit (see Fig. 6). As LSTM is a recurrent net, 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 Fig. 6, the output state vector for code token "try" is [1, -0.5, -3]. 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 $\mathcal{U} \in \mathbb{R}^{d' \times |\mathscr{V}|}$.

Our 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:

$$P(w_{t+1} = k \mid w_{1:t}) = \frac{\exp(\mathcal{U}_k^\top \boldsymbol{s}_t)}{\sum_{k'} \exp(\mathcal{U}_{k'}^\top \boldsymbol{s}_t)}. \quad (1)$$

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 kth of $\mathcal{U}^\top$, and k' runs through all the indices in the vocabulary, i.e., $k' \in \{1, 2, \ldots, |\mathscr{V}|\}$. This learning style essentially estimates a language model of code. In fact, our previous work [24] has demonstrated a good language model can be built based on LSTM, which suggest LSTM's capability to automatically learn a grammar for code [31].

Our LSTM is automatically trained using code sequences from all of the methods extracted from our dataset. During training, for every token in a sequence $\langle w_1, w_2, \ldots, w_n \rangle$, we know the true next token. For example, the true next token after "try" is "{" in the example Fig. 6. 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 $w_2, \ldots, -\log P(w_n \mid w_{1:n-1})$ 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:

$$L(\mathcal{P}) = -\log P(w_1) - \sum_{t=1}^{n-1} \log P(w_{t+1} \mid \boldsymbol{w}_{1:t}), \quad (2)$$

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

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.

5.2.3 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 [32]. 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. Code tokens that share common usage contexts in the corpus have their token semantics located in close proximity to one another in the space. Hence, the token states capture both syntactic and semantic of code tokens. The token states are thus used for generating two distinct sets of features for a file.

5.3 Generating Local Features

Generating local 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 $\langle w_1, w_2, \ldots, w_n \rangle$ from a method, feed it into the trained LSTM system, and obtain an output sequence of token states $\langle \boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_n \rangle$ (see Section 5.2.3). 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 Fig. 6). 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., $[\bar{\boldsymbol{s}}, \boldsymbol{\sigma}]$.

After the previous step, we obtain the method features for each method in a file. 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 local features for the file.

5.4 Generating Global Features

Local features are useful for within-project vulnerability prediction since they are capture the local usage context and thus tend to be project-specific. To enable effective cross-project vulnerability prediction, we need another set of features for a file which reflect how the file positions in a semantic space across all projects. We refer to these features as *global features*, similarly to the spirit of local and global models for defect prediction in [33].

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 challenge 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 [34]. 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.

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 (with Euclidian distance measure) to cluster all state vectors in the training set, where k is the pre-defined number of centroids, and hence the size of the

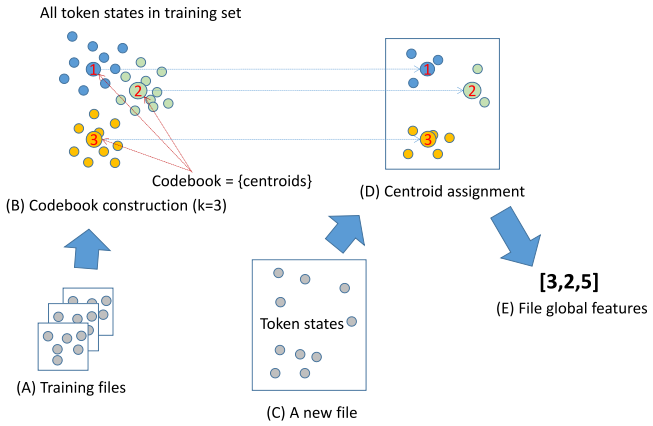

Fig. 7. An example of using "codebook" to automatically learn and generate global features of a new source file.

codebook (see Part A in Fig. 7, each small circle representing a token state in the space). The example in Fig. 7 uses $k = 3$ to produce three state clusters. For each new file, we obtain all the state vectors (Part B in Fig. 7) and assign each of them to the closest centroids (Part C in Fig. 7). The file is represented as a vector of length k, whose elements are the number of centroid occurrences. For example, the new file in Fig. 7 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 code tokens in a file. The intuition here is that the number of code 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, a 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. To the best of our knowledge, our work is the first to utilize the concepts of a "codebook" from computer vision to automatically learn and generate features for software code.

5.5 Vulnerability Prediction

The above process enables us to automatically generate both local and global features for all the source files in the training set. These files with their features and labels (i.e., vulnerable or clean) are then used to train machine learning classifiers. The trained model is used to predict the vulnerability of new files followed the standard process described in Section 3.1. Previous studies (e.g., [35]) conducted in industry have demonstrated that vulnerability prediction at the file level is indeed actionable. Although large source code files sometimes exist, the average file size is in the range of the hundreds of lines. Hence, inspecting a predicted source file to locate the exact locations of vulnerabilities is still feasible [35]. Although predictions at a finer level of granularity (e.g., line-level) potentially decrease manual inspection effort, they may well come with the cost of a severe reduction in accuracy. For example, the study in [35] shows that predicting vulnerabilities at the binary level (i.e., collections of source files) was more accurate than doing that at the file level. Predicting many false positives may reduce the usefulness of the machinery, and thus the user trust in the results. In addition, our approach learns features at the code token level. Those features can be aggregated to represent a line of code, or at a method and a whole source file level as we have demonstrated in this paper. We chose to apply our approach at the file level because almost all existing work and *datasets* in vulnerability prediction operates at this level of granularity. This facilitated us in training our models and performing comparisons.

To evaluate the impact of a classifier, we tested our approach with four widely-used classifiers: Random Forests, Decision Tree, Naive Bayes, and Logistic Regression. Random Forests has been shown to be an effective classifier for vulnerability prediction in previous studies [8], [26]. Random Forests belongs to a family of randomized ensemble methods which combines the estimates from many "weak classifiers" to make their prediction. Random Forests (RFs) [36] uses decision trees as weak learners. Those trees are trained using randomly sampled subsets of the full dataset. At each node of a decision tree, RFs find the best splitting feature (i.e., predictor) for the node from a randomly selected subset of features. For example, we might select a random set of 10 features (among 200 features) in each node, and then split using the best feature among these 10 features. Thus, RFs randomizes both training samples and feature selection to grow the trees. The same process is applied to generate more trees, each of which is trained using a slightly different sample each time. In practice, 100 to 500 trees are usually generated. To make predictions for a new instance, RFs combines all separate predictions made by each of the generated decision tree typically by averaging the outputs across all trees. Decision Tree (C4.5) generates decision nodes based on the information gain using the value of each factors. Decision tree classifier is widely used in practice due to its explainability. Naive Bayes works based the assumption that features are conditionally independent when the outcome is known. Despite of this naive assumption, Naive Bayes has been found to be an effective classifier. Logistic Regression uses the logistic sigmoid function to return a probability value for a given set of input features. This probability is then mapped into two or more discrete classes for classification purposes.

6 Implementation

The proposed approach is implemented in Theano [37] and Keras [38] frameworks, running in Python. Theano supports automatic differentiation of the loss in Eq. (2) and a host of powerful adaptive gradient descent methods. Keras is a wrapper making model building much easier.

6.1 Training Details

We use RMSprop as the optimizer and use the standard learning rate of 0.02, and smoothing hyper-parameters: $\rho = 0.99$, and $\epsilon = 1e - 7$. The model parameters are updated in a stochastic fashion, i.e., after every mini-batch of size 50. We use $|\mathcal{V}| = 5,000$ most frequent tokens for learning the code language model discussed in Section 5.2. We use *dropout* rate of 0.5 at the hidden output of LSTM layer. These parameter settings are the standard ones used in the literature.

The main classifier used is Random Forest implemented using the scikit-learn toolkit. Hyper-parameters are tuned for best performance and 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 @ 2.6 GHz. There machine has two CPUs, each has 8 physical cores or 16 threads, with a RAM of 128 GB.

6.2 Handling Large Vocabulary

To evaluate the prediction probability in Equation (1) 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 [39], 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].

The Noise Contrastive Estimation method replaces the expensive normalization through all tokens in the vocabulary by a simple logistic model on a small set of tokens. This ensures theoretically that the proper probability of each seen token is maintained given enough data. Hence, for large datasets, there is little loss in accuracy. This is a simple mathematical technique to improve the computation time of using an expensive normalization in a probabilistic model (e.g., the softmax), not a way to trade off quality for speed. Please note that we did not split long tokens into short ones.

6.3 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-batches 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.

7 Evaluation

7.1 Datasets

To carry out our empirical evaluation, we exploited two publicly available datasets that have been used in previous work for vulnerability prediction.

7.1.1 Android Dataset

This dataset [40] that has been used in previous work [8] 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 (i.e., clean or vulnerable files). 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 1
Dataset Statistics

App	#Versions	#Files	Mean files	Mean LOC	Mean Vuln	% Vuln
Crosswords	16	842	52	12,138	24	46
Contacts	6	787	131	39,492	40	31
Browser	6	433	72	23,615	27	37
Deskclock	6	127	21	4,384	10	47
Calendar	6	307	51	21,605	22	44
AnkiAndroid	6	275	45	21,234	27	59
Mms	6	865	144	35,988	54	37
Boardgamegeek	1	46	46	8,800	11	24
Gallery2	2	545	272	68,445	75	28
Connectbot	2	104	52	14,456	24	46
Quicksearchbox	5	605	121	15,580	26	22
Coolreader	12	423	35	14,708	17	49
Mustard	11	955	86	14,657	41	47
K9	19	2,660	140	50,447	65	47
Camera	6	457	76	16,337	29	38
Fbreader	13	3,450	265	32,545	78	30
Email	6	840	140	51,449	75	54
Keepassdroid	12	1,580	131	14,827	51	39

We could not find the code repository for two applications since they appeared no longer available[1]. 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 were able to 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.

Our resultant dataset contains applications from two sources: 9 applications from F-Droid repository and 9 applications pre-installed with Android OS. All the nine F-Droid applications had over 10,000 downloads, and five of them had more than 1 million downloads. There were 21 types of vulnerabilities existing across all of the applications in the dataset, such as log forging, information leak, unreleased resource, denial of service, race condition, cross-site scripting, command injection, privacy violation and header manipulation. Among them, privacy violation, log forging and denial of service are the top three common vulnerabilities found in the dataset. Table 1 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. 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 the validation set was finally kept for feature extraction.

1. We had tried to contact the owners of the original dataset, but did not receive any response. We also contacted the owners of some applications in the dataset. Some of them helped us locate the source code of the correct versions.

7.1.2 Firefox Dataset

The Firefox vulnerabilitity dataset was previously built by Shin and Williams [12]. They collected vulnerabilities reported for Firefox 2.0 from the Mozilla Foundation Security Advisories (MFSAs). They linked these MFSA reports with bug reports in Firefox, and used this information to identify the files in Firefox 2.0 that contain the reported vulnerabilities. The dataset has all 11,051 source files, all of them from Firefox 2.0. There are 363 vulnerable files, accounting for 3 percent of the total files. From the dataset they provided, we retreived the code of those source files from the archived Firefox code repository.

7.2 Research Questions

We followed previous work in vulnerability prediction [8] and aimed to answer the following standard research questions:

1) *RQ1. Within-project prediction*: *Are the automatically learned features using our LSTM-based approach suitable for building a useful vulnerability prediction model?*
2) *RQ2. Cross-version prediction*: *How does our proposed approach perform in predicting future versions, i.e., when the model is trained using an older version in application and tested on a newer version in the same application?*
3) *RQ3. Cross-project prediction*: *Is our approach suitable for cross-project predictions where the model is trained using an application and tested on a different application?*

7.3 Experimental Settings

We designed three different experimental settings to answer the above research questions.

7.3.1 Within-Project Prediction

In this setting, both training and testing data is from the same version of an application. Specifically, for each application in our dataset, we select all the source files in the first version, and use them for training and testing a prediction model. We employ stratified cross-fold validation by dividing the files into 10 folds $\{fo_1, fo_2, \ldots, fo_{10}\}$, each of which has the approximately same ratio between vulnerable files and clean files. For each fold fo_i, we select it as the test set, and the remaining folds $\{fo_1, \ldots, fo_{i-1}, fo_{i+1}, \ldots, fo_{10}\}$ are used for training a prediction model. We then measure the performance of the prediction model using the source files in the fold fo_i. We repeat this process for each of the 10 folds, and then compute the average performance.

7.3.2 Cross-Version Prediction

In this second experimental setting, a prediction model is trained using all the source files in one version of an application. It is then tested using the source files from 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.

We also experimented with another setting in which the model training is updated over time. Specifically, the model is trained using all of the previous releases rather than only the first one. For example, if an application has 10 versions, we conduct 9 different runs. In first run, the model is trained using the version 1 and tested using the remaining 9 versions. In the second run, the model is trained using versions 1 and 2, and tested using the remaining 8 versions. We repeat this process and compute the average performance.

7.3.3 Cross-Project Prediction

In this third experiment, we used all the source files in the first version of each application to train a prediction model. The model is then tested on the first version of the remaining 17 applications, i.e., it is tested 17 times. We compute the performance in each test and use the average performance. We repeat this procedure for all applications, resulting in 18 different prediction models, one for each application used for training.

7.4 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., [41]) and vulnerability prediction (e.g., [9], [10], [11]). 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 (i.e., measure to what extent the source code elements are coherent in the system), complexity metrics (i.e., measure the complexity of source code elements such as algorithms), coupling metrics (i.e., measure the amount of interdependencies of source code elements), documentation metrics (i.e., measure the amount of comments and documentation of source code elements), inheritance metrics (i.e., measure the different aspects of the inheritance hierarchy), code duplication metrics (i.e.), and size metrics (e.g., number of code lines, and number of classes or methods).

Bag of Words. This technique has been used in previous work [8], which also considers source code as a special form of text. Hence, it treats a source code 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 [8] 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 [15] has demonstrated that Deep Belief Networks (DBN) [42] work well for defect prediction. DBN is a family of stochastic deep neural networks that 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. Unlike the standard neural networks, which learn a mapping from an input to an output in an supervised manner, RBM learns a distribution of data using unsupervised learning (i.e., without labels). Following the standard practice in the literature, the RBM is trained using Contrastive Divergence [42]. 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 has been shown to be one of the most effective classifiers for vulnerability prediction [8]. We used the implementation of RF provided with the scikit-learn[2] toolkit. We performed tuning with the following hyper-parameters: the number of trees, the maximum depth of a tree, the minimum number of samples required to split an internal node, and the maximum number of features. The same hyper-parameter tuning was done for our prediction models and all the benchmarks.

7.5 Performance Measures

A confusion matrix is used to evaluate the performance of our predictive models. The confusion matrix is then used to store the correct and incorrect decisions made by a classifier. For example, if a file is classified as vulnerable when it was truly vulnerable, the classification is a true positive (*tp*). If the file is classified as vulnerable when actually it was not vulnerable, then the classification is a false positive (*fp*). If the file is classified as clean when it was in fact vulnerable, then the classification is a false negative (*fn*). Finally, if the file is classified as clean and it was in fact clean, then the classification is true negative (*tn*). The values stored in the confusion matrix are used to compute the widely-used Precision, Recall, and F-measure for the vulnerable files. These measures have also been widely used in the literature (e.g., [5], [8], [10], [11]) for evaluating the predictive performance of vulnerability prediction models.

- Precision (Prec): The ratio of correctly predicted delayed issue over all the issues predicted as delayed issue. It is calculated as:

$$pr = \frac{tp}{tp + fp}.$$

- Recall (Re): The ratio of correctly predicted delayed issue over all of the actually issue delay. It is calculated as:

$$re = \frac{tp}{tp + fn}.$$

- F-measure: Measures the weighted harmonic mean of the precision and recall. It is calculated as:

$$F - measure = \frac{2 * pr * re}{pr + re}.$$

- Area Under the ROC Curve (AUC) is used to evaluate the degree of discrimination achieved by the model. The value of AUC is ranged from 0 to 1 and random prediction has AUC of 0.5. The advantage of AUC is that it is insensitive to decision threshold like precision and recall. The higher AUC indicates a better predictor.

7.6 Results

7.6.1 Learned Code Token Semantics

An important part of our approach is learning the semantics of code tokens using the context of its usage through a LSTM approach. Fig. 8 shows the top 2,000[3] frequent code tokens used in our dataset. They were automatically grouped in 100 clusters (only 10 are shown in Fig. 8) 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 5.4). We used t-distributed stochastic neighbor embedding (t-SNE) [43] 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 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 later learn both syntactic and semantic features.

7.6.2 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 local and global features. We report here the results of using Random Forests as the classifier (the same for RQ2 and RQ3). For space reasons, the detailed results of using Decision Tree, Naive Bayes, and Logistic Regression are reported in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2018.2881961. Table 2 reports the precision, recall, F-measure and AUC for the three variations of our approach: using only local features, using only global features, and using a joint set of both features types. Note that both the local feature option and the joint feature option we reported here used mean pooling for generating method features and standard deviation pooling for generating syntactic file features.

Our approach obtained a strong result: in *all* 18 Android applications, they all achieved well above 80 percent in across all the four measures: precision, recall, F-measure, and AUC (see Table 2). Among the three variations of our approach, using only local features appeared to be the best option for

2. http://scikit-learn.org

3. We choose to show the top 2,000 frequent code tokens used in our dataset as this number of code tokens allows us to easily see how they are grouped into clusters in Fig. 8. We could display a significantly larger number of code tokens in the figure, but it would affect the visibility of the figure.

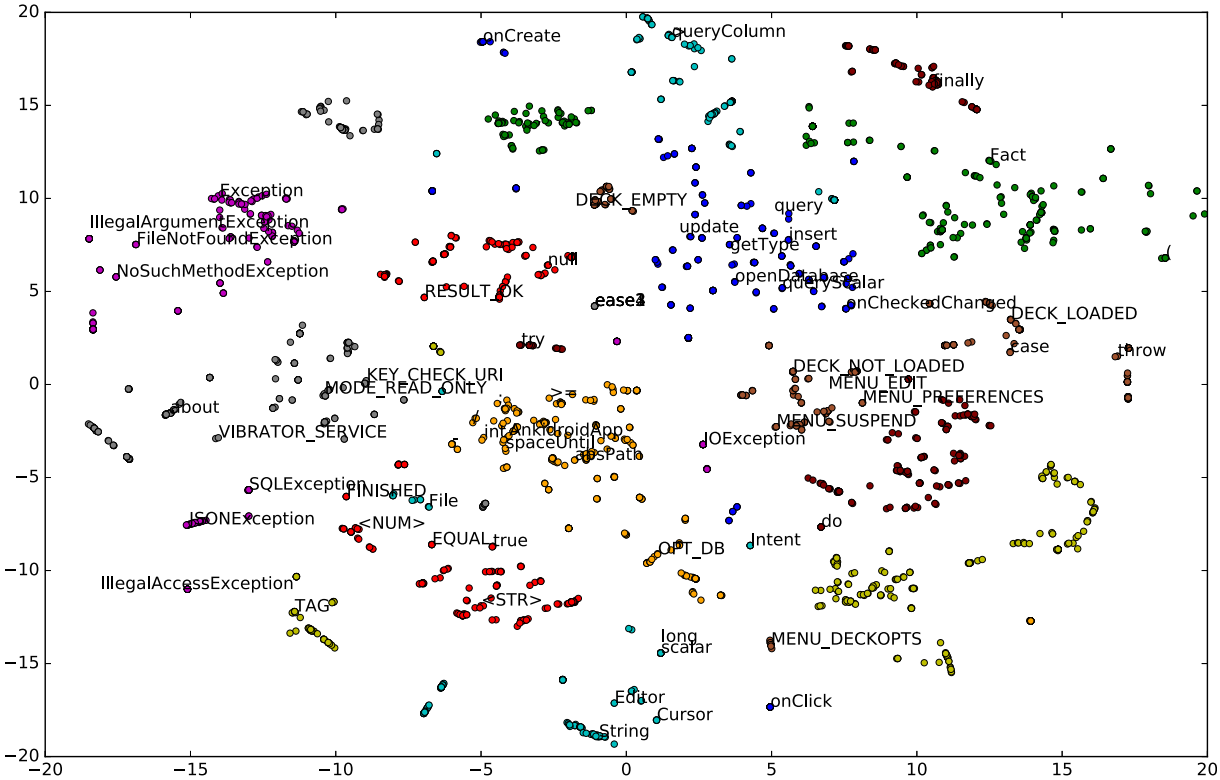

Fig. 8. Top 2,000 frequent code tokens were automatically grouped into clusters (each cluster has a distinct color).

TABLE 2
Within-Project Results (RQ1) for the Three Variations of Our Approach

Application	Local features				Global features				Joint features			
	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
AnkiAndroid	1.00	1.00	1.00	1.00	1.00	0.94	0.96	1.00	1.00	0.94	0.96	1.00
Boardgamegeek	0.95	0.95	0.93	0.95	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90
Browser	1.00	1.00	1.00	1.00	0.96	1.00	0.98	0.92	0.96	1.00	0.98	0.95
Calendar	1.00	1.00	1.00	0.96	0.96	0.94	0.94	0.97	1.00	1.00	1.00	1.00
Camera	0.88	1.00	0.93	0.94	0.92	0.98	0.94	0.93	0.92	0.98	0.94	0.92
Connectbot	1.00	0.97	0.98	0.96	0.93	0.93	0.93	0.97	0.97	0.93	0.93	0.95
Contacts	0.96	0.94	0.95	0.90	0.89	0.94	0.89	0.92	0.87	0.97	0.90	0.90
Coolreader	1.00	0.94	0.96	0.97	1.00	0.94	0.96	0.96	1.00	0.94	0.96	0.96
Crosswords	0.89	0.89	0.89	0.96	0.89	0.83	0.85	0.86	0.89	0.83	0.85	0.91
Deskclock	1.00	1.00	1.00	1.00	1.00	0.92	0.94	0.90	1.00	0.92	0.94	0.84
Email	0.93	0.95	0.93	0.93	0.95	0.90	0.92	0.93	0.94	0.93	0.93	0.96
Fbreader	0.83	0.82	0.82	0.96	0.85	0.83	0.84	0.90	0.82	0.86	0.83	0.95
Gallery2	0.83	0.88	0.84	1.00	0.83	0.87	0.84	0.94	0.80	0.89	0.83	0.95
K9	0.89	1.00	0.94	0.92	0.88	0.97	0.91	0.92	0.88	0.98	0.92	0.95
Keepassdroid	0.93	0.97	0.94	0.95	0.90	0.91	0.89	0.90	0.88	0.89	0.88	0.89
Mms	0.87	0.85	0.85	0.92	0.87	0.88	0.87	0.94	0.87	0.92	0.89	0.95
Mustard	0.96	0.99	0.97	1.00	0.97	0.93	0.94	0.93	0.95	0.99	0.96	0.98
Quicksearchbox	0.93	0.90	0.88	0.98	0.89	0.77	0.82	0.88	0.89	0.79	0.82	0.93
Firefox	0.33	0.15	0.20	0.57	0.30	0.49	0.37	0.89	0.28	0.48	0.35	0.88

"Joint features" indicates the use of both local features and global features.

vulnerability prediction. This result is consistent with our underlying theory that local features are project-specific and are thus useful for within-project prediction. This option delivered the highest precision, recall, F-measure, and AUC averaging across 18 applications. The prediction model using local features achieved over 90 percent F-measure in 13 out of 18 applications and over 90 percent AUC in all 18 applications.

Our approach outperforms the three benchmarks to varying extents (see Fig. 9). Fig. 9 shows an number of box plots to demonstrate the performance improvement of our approach over an existing technique. Each box plot shows the distribution of the improvement across the seventeen projects used in our dataset (e.g., AnkiAndroid, Boardgamegeek, etc.) for a specific performance indicator (e.g., F-measure). The bottom and top of a box are the first and third quartiles, and the band

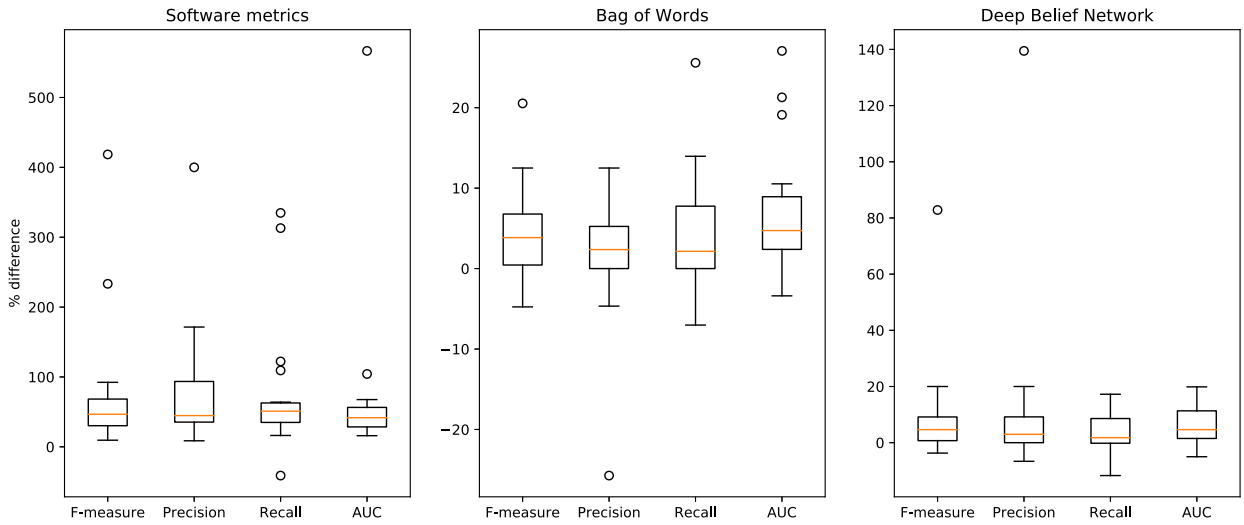

Fig. 9. The percentage performance difference when applying the three benchmarks (software metrics, Bag of Words and Deep Belief Network) against using our local feature approach for each performance measure in within-project prediction.

inside the box indicate the second quartile (i.e., the median). The lines extending parallel from the boxes indicate variability outside the upper and lower quartiles, and individual dots indicate outliers. For example, the first box plot on the left-hand-side of Fig. 9 shows the improvement distributed across the seventeen projects of our model over the software metrics approach in terms of F-measure. As can also be seen in Fig. 9, across the 17 projects, the median improvement is approximately 45 percent (refer to Appendix C, available in the online supplemental material, for the distribution of the absolute values of a given performance measure).

For all 18 Android applications, our local feature approach consistently outperformed the software metric approach in all performance measures. For over half of the applications, our approach offered over 50 percent improvement in F-measure over the software metrics approach. In some applications (e.g., Boardgamegeek and Deskclock), the improvements were over 200 percent. The Bag-of-Words (BoW) and Deep Belief Network approaches also achieved good results (on average 89–90 percent F-measure). This is consistent with the findings in previous work [8] in the case of BoW. Hence, the improvements brought by our approach over these two benchmarks were not as big as those over the software metric approach (see Fig. 9). Our approach outperformed the BoW approach in 15 out of 18 applications, and in most cases the improvements ranged from 4 to 20 percent. The results show that Random Forests clearly outperform Decision Tree, Logistic Regression and Naive Bayes for within-project vulnerability prediction in all three settings: using local features, global features, and both of them. The other three classifiers (Logistic Regression, Naive Bayes and Decision Tree) produced mixed results (see Figs. 12, 13, and 14 in Appendix A, available in the online supplemental material). The improvement brought by our approach over BoW and DBN was not as clear for using those classifiers as it was for using Random Forests.

For the Firefox application, the dataset is highly unbalanced[4] in that only 3 percent of the 11,051 files are vulnerable (i.e., minor class). Thus, a classification model tends to have a bias towards predicting files as being clean (i.e. major class) in order to reducing training errors. This may result in the vulnerable class having high false negatives. To deal with this issue, previous studies (e.g., [12]) often employ class rebalancing techniques (e.g., oversampling or undersampling) to make the training set balanced. Those resampling techniques however create an artificial bias towards minor classes. An alternative is imposing an additional cost on the model for making classification mistakes on the minor class during training (i.e., penalized classification). These costs enable the model to pay more attention to the minority class. We have conducted experiments using undersampling, oversampling and penalized classification. The results of using penalized classification are reported here, while the results of using undersampling and oversampling techniques are reported in Appendix D, available in the online supplemental material.

Similarly to previous study [12] on this Firefox dataset, we have performed a sensitivity analysis on the classification threshold in which we evaluated our prediction models at 91 classification thresholds, ranging from 0.01 to 0.91 at an interval of 0.01 (instead of selecting a single default threshold). The classification threshold is used by the model to classify files based on the predicted probabilities.

Overall, the models using global features and joint features outperformed the model using local features (see Fig. 10). Using global or joint features consistently achieved well above 0.8 AUC for all classification thresholds while the AUC produced by the model using local features is below 0.6. Using local features produced high recall but low precision at the threshold up to 0.45, but recall dramatically decreases and precision rapidly increases when the threshold is set above 0.45. The model using global or joint features produced a more balance between precision and recall although there are also regions where recall or precision is optimized and vice versa. If the security engineers are reluctant to take risks and they have sufficient resources for testing and inspection, it is recommended that they select a model which has a good performance at a low threshold. On the other hand, if they have limited resource for testing and inspection, they can select a model which produces high precision at a high threshold. Compared to the model developed

4. Unbalanced data is not uncommon in vulnerability prediction.

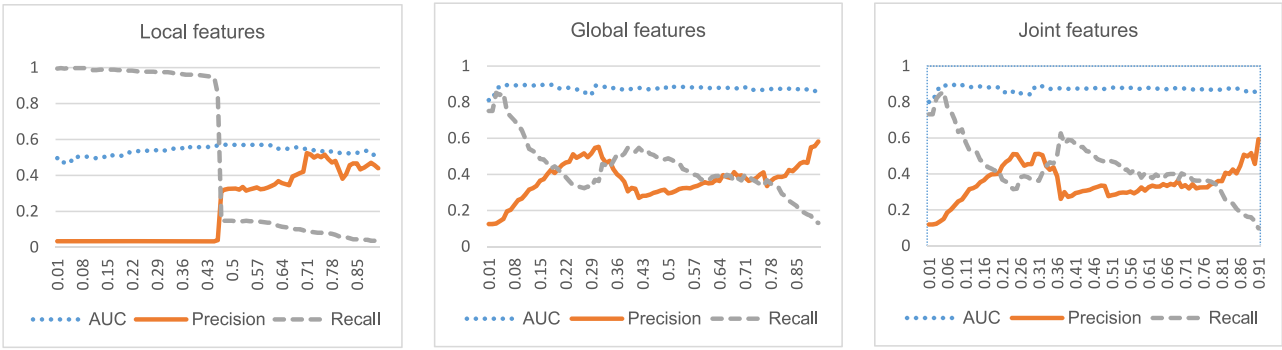

Fig. 10. Within-project results (RQ1) for the three variations of our approach on the Firefox dataset with classification threshold varying from 0.01 to 0.91. "Joint features" indicates the use of both local features and global features.

TABLE 3
Cross-Version Results (RQ2) for the Three Variations of Our Approach

Application	Local features				Global features				Joint features			
	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
AnkiAndroid	0.83	0.90	0.87	0.90	0.91	0.84	0.87	0.90	0.89	0.87	0.88	0.91
Browser	0.59	0.64	0.61	0.84	0.84	0.57	0.68	0.83	0.82	0.58	0.68	0.72
Calendar	0.73	0.81	0.77	0.87	0.79	0.89	0.83	0.89	0.79	0.84	0.82	0.83
Camera	0.65	0.84	0.73	0.86	0.67	0.88	0.76	0.92	0.84	0.80	0.82	0.90
Connectbot	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Contacts	0.60	0.80	0.69	0.88	0.69	0.72	0.70	0.86	0.73	0.69	0.71	0.85
Coolreader	0.94	0.89	0.91	0.93	0.89	0.85	0.87	0.93	0.92	0.86	0.89	0.94
Crosswords	0.85	0.89	0.87	0.91	0.88	0.90	0.89	0.95	0.92	0.83	0.87	0.94
Deskclock	0.93	0.80	0.86	0.90	0.90	0.86	0.88	0.89	0.89	0.82	0.86	0.94
Email	0.77	0.93	0.84	0.87	0.86	0.89	0.87	0.91	0.83	0.90	0.86	0.90
Fbreader	0.84	0.81	0.83	0.93	0.91	0.84	0.88	0.96	0.86	0.84	0.85	0.95
Gallery2	0.95	0.97	0.96	0.98	0.96	0.99	0.97	1.00	0.96	0.97	0.97	1.00
K9	0.80	0.94	0.87	0.88	0.84	0.98	0.90	0.96	0.89	0.92	0.90	0.93
Keepassdroid	1.00	0.99	0.99	0.94	0.98	0.91	0.94	0.97	0.99	0.92	0.95	1.00
Mms	0.93	0.89	0.91	0.95	0.93	0.93	0.93	0.97	0.92	0.96	0.94	0.97
Mustard	0.98	0.98	0.98	0.98	0.99	0.95	0.97	0.99	0.99	0.95	0.97	1.00
Quicksearchbox	0.76	0.80	0.78	0.91	0.74	0.87	0.80	0.95	0.77	0.88	0.82	0.94
Average	**0.83**	**0.87**	**0.85**	**0.92**	**0.87**	**0.87**	**0.87**	**0.93**	**0.88**	**0.86**	**0.87**	**0.94**

"Joint features" indicates the use of both local features and global features.

by Shin and Williams [12], our models using joint or global features consistently produced higher precision at all the thresholds. For example, at threshold 0.91, our model using joint features achieved 0.60 percent precision, offering 15 percent improvement over the model developed by Shin and Williams [12], while the improvement was 130 percent at threshold 0.5. Our model also produced a better balance between precision and recall, resulting in higher F-measure at most of the thresholds compared to the Shin and Williams' models.

Answer to RQ1: Features automatically learned from source code using LSTM can be used to build highly accurate prediction models for detecting vulnerabilities in software applications.

7.6.3 Cross-Version Prediction (RQ2)

Table 3 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. Note that the Boardgamegeek application had only one version and thus was excluded from this experiment. Hence, there were 17 application evaluated in this experiment. The results demonstrate that the three variations of our approach again achieved strong predictive performance in all criteria. Average precision, recall and F-measure values are approximately 85 percent, while average AUC values are above 90 percent.

The use of global features has improved the generalization of the prediction models in the cross-version setting. Using both local features and global features appeared to be the best option in this setting. This approach has achieved well above 80 percent in all the four performance measures in 14 out of 17 applications. Our approach (using joint features) outperformed the software metrics approach with respect to all performance measures in all 17 applications (see Fig. 21 and refer to Appendix C, available in the online supplemental material, for the distribution of the absolute values of a given performance measure). The average improvement over the software metric approach is 15 percent for F-measure and 20 percent for AUC.

Our approach outperformed the BoW approach in 16 out of 17 cases in terms of F-measure and AUC, and the DBN approach in 15 out of 17 cases in F-measure (and 16

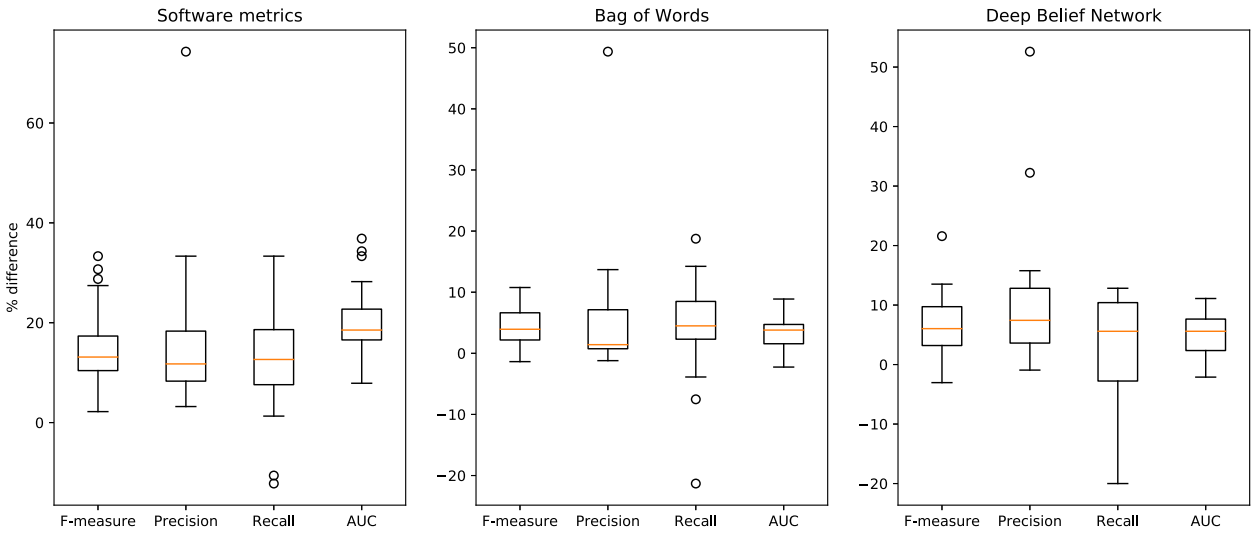

Fig. 11. The percentage performance difference when applying the three benchmarks (software metrics, Bag of Words and Deep Belief Network) against using our local feature approach for each performance measure in cross-version prediction.

cases for AUC) see Fig. 11. Note that the BoW and DBN approaches also performed well in cross-version prediction, i.e. their average F-measure are above 80 percent. In some cases (e.g., the Camera application), they achieved high recall (e.g., 88 percent for DBN and 90 percent for BoW) but low precision (e.g., 58 and 59 percent). In those cases, our approach achieved a more balance performance between recall and precision. For example, in the Camera application, our approach achieved lower recall (i.e. 71 percent) than BoW and DBN did, but it produced a higher precision (i.e. 79 percent), and thus lead to 10–12 percent improvement in F-measure and 8–10 percent improvement in AUC. The results show that Random Forests clearly outperform Decision Tree, Logistic Regression and Naive Bayes for cross-version vulnerability prediction in all three settings: using local features, global features, and both of them. When using Logistic Regression and Decision Tree as the classifier, similar improvements were also observed (see Figs. 15 and 16 in Appendix A, available in the online supplemental material). The improvement brought by our approach over BoW and DBN was not as clear for Naive Bayes (see Fig. 17 in Appendix A, available in the online supplemental material) as it was for using the other three classifiers.

Table 4 shows the results in the setting which the model training is updated over time. Here, the model was trained using using all the previous releases (rather than only the first one). The results were averaged of all runs in each project. The results are slightly improved, suggesting that updating model training over time is helpful.

Answer to RQ2: Our predictive model, which is trained using an older version of a software application, can produce highly accurate predictions of vulnerable components in the new versions of the same application.

TABLE 4
Cross-Version Results (RQ2) for the Three Variations of Our Approach, Using Random Forest in the Updated Training Setting

Application	Local features				Global features				Joint features			
	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
Ankiandroid	0.84	0.92	0.88	0.91	0.86	0.91	0.88	0.9	0.87	0.91	0.89	0.92
Browser	0.72	0.64	0.67	0.78	0.81	0.6	0.69	0.83	0.83	0.59	0.68	0.83
Calendar	0.78	0.8	0.79	0.84	0.78	0.87	0.82	0.89	0.79	0.83	0.8	0.89
Camera	0.8	0.8	0.79	0.9	0.82	0.79	0.8	0.9	0.87	0.78	0.82	0.9
Connectbot	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Contacts	0.62	0.75	0.68	0.85	0.69	0.77	0.73	0.89	0.66	0.76	0.71	0.85
Coolreader	0.95	0.89	0.92	0.97	0.95	0.87	0.91	0.95	0.96	0.87	0.91	0.95
Crosswords	0.86	0.94	0.9	0.94	0.88	0.93	0.91	0.94	0.88	0.94	0.91	0.95
Deskclock	0.92	0.85	0.88	0.94	0.92	0.88	0.9	0.92	0.92	0.87	0.89	0.92
Email	0.79	0.94	0.86	0.9	0.85	0.9	0.87	0.92	0.83	0.91	0.86	0.91
Fbreader	0.91	0.87	0.89	0.97	0.93	0.89	0.91	0.97	0.93	0.88	0.9	0.97
Gallery2	0.95	0.97	0.96	0.99	0.96	0.96	0.96	0.99	0.96	0.96	0.96	0.99
K9	0.89	0.96	0.92	0.96	0.91	0.97	0.94	0.97	0.9	0.97	0.93	0.97
Keepassdroid	1.0	0.98	0.99	0.99	0.99	0.94	0.97	0.99	0.99	0.94	0.97	0.99
Mms	0.96	0.95	0.95	0.98	0.97	0.94	0.96	0.97	0.97	0.95	0.96	0.97
Mustard	0.98	0.98	0.98	0.99	0.99	0.96	0.97	0.99	0.99	0.96	0.98	0.99
Quicksearchbox	0.85	0.84	0.85	0.96	0.8	0.92	0.85	0.97	0.88	0.88	0.88	0.97
Average	**0.87**	**0.89**	**0.88**	**0.93**	**0.89**	**0.89**	**0.89**	**0.94**	**0.9**	**0.88**	**0.89**	**0.94**

"Joint features" indicates the use of both local features and global features.

TABLE 5
Cross-Project Results (RQ3) for the Three Benchmarks and Three Variations of Our Approach

App	Metrics	BoW	DBN	Local	Global	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**

Numbers are the number of other applications to which learned models can be applied.

7.6.4 Cross-Project Prediction (RQ3)

This experiment followed the setup in previous work [8]. 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. Hence, for each prediction method, there are 18 x 17 (i.e. 306) different settings. We ran this experiment with the three benchmarks and variations of our approach. In this experiment, we do not focus on the raw performance in each setting. Instead, we follow the same procedure as previously done in our benchmark [8], and focus on assessing how many applications a model can be effectively applied to. We used the same baseline as in previous work [8]: a model is applicable to a tested application if both precision and recall are above 80 percent.

For each application, Table 5 reports the number of other applications to which the corresponding models can be applied. The results show that using global features improves 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, approximately doubling the number of applications achieved by BoW or DBN models. The results also show that Random Forests clearly outperform Decision Tree, Logistic Regression and Naive Bayes for cross-project vulnerability prediction in all three settings: using local features, global features, and both of them. In addition, our approach also outperforms BoW and DBN when using Decision Tree and Naive Bayes as the classifier (see Tables 12 and 14 in Appendix A, available in the online supplemental material), but this was not the case when using Logistic Regression as the classifier (see Table 13 in Appendix A, available in the online supplemental material).

Cross-project prediction is always challenging due to the potentially significant differences (e.g., coding styles and functionalities) between projects. Although the result is encouraging, we however acknowledge that further work is needed to improve this result to make it more applicable in practice. One potential improvement is learning the features from a set of diverse applications (rather than one single application).

Answer to RQ3: Some predictive models, which were trained and used features automatically learnt from a software application, can predict vulnerable software components in other software applications.

7.7 Discussion

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 representing an executable programming language. The other reason is that although the training files and testing 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 used can come from multiple forms: fully repeated pieces of code, the same BoW statistics, 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.

Finally, almost all existing work (e.g., [5], [8], [9], [10], [11], [26], [27], [28]) in vulnerability prediction operates at the file level. Since this type of prediction (i.e. with the file granularity) is standard in the related work, we chose to work at the file level to leverage existing datasets and facilitate comparison against state-of-the-arts. However, our approach is able to learn features at the code token level, and thus it may work beyond the file granularity. In fact, since we consider a method as a sequence of code tokens, our current model is already able to automatically learn and generate features for the method. These features can be used to build a model for predicting vulnerabilities at the method level (discussed in Section 5). In the same

manner, we can treat each line of code as a sequence of code tokens and use aggregation to obtain features for each code line. Thus, our approach is also potentially applicable to vulnerability prediction at line level. Training a prediction model at those levels of granularity requires corresponding ground-truths, i.e. methods or code lines which have been labelled as vulnerable or clean. The existing vulnerability datasets (like the one we used from [8] and others used in previous studies) unfortunately do not contain labels at the method or code line levels. Once such datasets become available, our model is readily extensible to leverage them. Alternatively, some recent studies in the general defect prediction area target at the method level (e.g., [44]) and the line level (e.g., [45]). Hence, another possibility, which we will investigate in our future work, is extending our model to general defect prediction and making use of those datasets.

8 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 [8]. The dataset contains real 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. We acknowledge that this dataset may contain false positives and one approach to deal with this is manually removing them from the dataset. However, the dataset from [8] does not provide detailed reports of the vulnerabilities, and thus we were unable to remove the false positives. However, as reported in [8], they have removed the false positives in two applications (AnkiDroid and Mustard) and found that it did not negatively affect the result in these two cases. This may suggest that the performance of our approach would not also be negatively affected by removing false positives from the dataset. In addition, as can be seen in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/2881961, our models also produce highly accurate predictions of clean components (i.e. negative cases) in all three settings (within-project, cross-version and cross-projects).

Conclusion Validity. We tried to minimize threats to conclusion validity by using standard performance measures for vulnerability prediction [8], [10], [11], [26]. We however acknowledge that a number of statistical tests [46] 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. The benchmarks (Software Metrics, Bag of Words, Deep Belief Network) are techniques that have been proposed and implemented in existing work (e.g., [8], [15]). However, their original implementations (e.g., the tools) were not made publicly available, and thus we were not able to have access to them. Hence, we had to re-implement our own version of those techniques. Although we closely followed the procedures described in their work, we acknowledge that our implementation might not have all of the details, particularly those not explicitly described in their papers, of the original implementation. However, to ensure that our implementation reflects the original implementation, we tested our implementation using the same dataset, and our implementation produces similar and consistent results.

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 [8] 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 software applications. Further investigation to confirm our findings for other types of applications such as web applications and applications written in other programming languages such as PHP and C++.

9 Related Work

9.1 Vulnerability Prediction

Machine learning techniques have been widely used to build vulnerability prediction models. Early approaches (e.g., [9]) 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., [11]), code churn and developer activity metrics (e.g., [10])), 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 [8] 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. BoW models produced higher recall than software metric models for PHP applications [26]. A recent case study [27] for the Linux Kernel also suggested the superiority of BoW in vulnerability prediction compared to using code metrics as features. However, the BoW approach carries the inherent limitation of BoW in which syntactic information such as code order is disregarded. These approaches also rely on manual feature engineering in which discriminative features are extracted by a deliberate process of combining primitive components such as tokens. Examples of such features are n-grams, topics, special selection of keywords, types of expression, etc. These features are often sufficient for shallow models such as Naive Bayes, SVM and Random Forests to perform well. In deep learning, tokens are used as the starting point because they are the smallest unit readily available in code (we can argue that characters are the smallest, but tokens are easier to reason about). No further domain knowledge is assumed other than the fact that code is

a sequence of tokens. The higher level features are learnt automatically by estimating parameteric neurons at multiple levels of abstraction.

9.2 Defect Prediction

Predicting vulnerabilities is related to software defect prediction, which is a very active area in software analytics. Since defect prediction is a broad area, we highlight some of the major work here, and refer the readers to other comprehensive reviews (e.g., [47], [48]) for more details. Code metrics were commonly used as features for building defect prediction models (e.g., [41]). Various other metrics have also been employed such as change-related metrics [49], [50], developer-related metrics [51], organization metrics [52], and change process metrics [53]. The study in [12] found that some defect prediction models can be adapted for vulnerability prediction. However, most of those models are not directly transferred to predicting secutiry vulnerabilities [8].

Recently, a number of approaches (e.g., [15], [54]) have leveraged a deep learning model called Deep Belief Network (DBN) [55] to automatically learn features for defect prediction and have demonstrated an improvement in predictive performance. DBN however does not naturally capture the sequential order and long-term dependencies in source code. Most of the studies in defect prediction operate at the file level. Recent approaches address this issue at the method level (e.g., [44]) and the line level (e.g., [45]). Since our approach is able to learn features at the code token level, it may work at those finer levels of granularity. However, this would require the development of a vulnerability dataset for training that contains methods and codelines with vulnerability labels, which do not currently exist.

9.3 Deep Learning in Code Modeling

Deep learning has recently attracted increasing interests in software engineering. In our recent vision paper [56], we have proposed DeepSoft, a generic deep learning framework based on LSTM for modeling both software and its development and evolution process. We have demonstrated how LSTM is leveraged to learn long-term temporal dependencies that occur in software evolution and how such deep learned patterns can be used to address a range of challenging software engineering problems ranging from requirements to maintenance. Our current work realizes one of those visions.

The work in [17] demonstrated the effectiveness of using recurrent neural networks (RNN) to model source code. Their later work [57] extended these RNN models for detecting code clones. The work in [58] 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 [59] also uses RNN Encoder–Decoder but for fixing common errors in C programs. The work in [60] uses Convolutional Neural Networks (CNN) [61] for bug localization. Preliminary results from our earlier work [24] also suggest that LSTM is a more effective language model for source code. Our work is built on this language model to automatically learn both syntactic and semantic features for predicting vulnerable code components.

10 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 [8]. The results for within-project prediction demonstrate that our approach achieved well above 80 percent in all performance measures (precision, recall, F-measure, and AUC) in all 18 Android applications. When using Random Forests as the classifier, our approach also outperforms the traditional software metrics approach (74 percent improvement on average), the Bag-of-Words approach (4.5 percent improvement on average) and another deep learning approach, Deep Belief Network (5.2 percent improvement on average). For cross-project prediction, the results suggest that a predictive model, which was trained from an Android application using our approach, can predict vulnerable software components in (on average) 4 other Android applications with both precision and recall above 80 percent – doubling the number of applications achieved by either Bag-of-Words or Deep Belief Network. An evaluation on the Firefox application also demontrates that our models improved from 23 to 175 percent in precision compared to existing models.

Our future work involves applying this 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.

Acknowledgments

The authors gratefully acknowledge support from Samsung through its 2016 Global Research Outreach Program. We would also thank Yonghee Shin for sharing with us the Firefox vulnerability dataset.

References

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

[2] McAfee, C. for Strategic, and I. Studies, "Net Losses: Estimating the Global Cost of Cybercrime," Jun. 2014. [Online]. Available: https://www.sbs.ox.ac.uk/cybersecurity-capacity/system/files/McAfee%20and%20CSIS%20-%20Econ%20Cybercrime.pdf

[3] C. Ventures, "Cybersecurity market report," Mar. 2017. [Online]. Available: http://cybersecurityventures.com/cybersecurity-market-report, Accessed On: May 01, 2017.

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

[5] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation*, 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 *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 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 *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2016, pp. 25:1–25:6. [Online]. Available: http://doi.acm.org/10.1145/2961111.2962611

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

[9] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2008, pp. 315–317. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414065

[10] 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

[11] 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

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

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

[14] K. Z. Sultana, "Towards a software vulnerability prediction model using traceable code patterns and software metrics," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 1022–1025. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155700

[15] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[16] 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 *Proc. 7th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2009, pp. 91–100.

[17] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345.

[18] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[19] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[20] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847.

[21] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2005, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081755

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

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

[24] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," in *Proc. Workshop Naturalness Softw. (NL+SE), co-located with 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 1–4.

[25] D. Mohindra, "SEI CERT Oracle Coding Standard for Java," [Online]. Available: https://www.securecoding.cert.org/confluence/display/java/LCK08-J.+Ensure+ac

[26] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*,, 2014, pp. 23–33. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2014.32

[27] M. Jimenez, M. Papadakis, and Y. L. Traon, "Vulnerability prediction models: A case study on the linux kernel," in *16th IEEE Int. Work. Conf. Source Code Anal. and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2–3, 2016*, 2016, pp. 1–10. [Online]. Available: https://doi.org/10.1109/SCAM.2016.15

[28] F. Massacci and V. H. Nguyen, "An empirical methodology to evaluate vulnerability discovery models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 12, pp. 1147–1162, Dec. 2014.

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

[30] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Field Guide to Dynamical Recurrent Networks," *IEEE Press*, 2001.

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

[32] M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics*, 2014, pp. 238–247.

[33] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 822–834, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.83

[34] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. 7th IEEE Int. Conf. Comput. Vis.*, 1999, vol. 2, pp. 1150–1157.

[35] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Security*, 2015, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2746194.2746198

[36] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

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

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

[39] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *J. Mach. Learn. Res.*, vol. 13, pp. 307–361, 2012.

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

[41] 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

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

[43] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 2579–2605, 2008, Art. no. 85.

[44] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2012, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/2372251.2372285

[45] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

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

[47] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4–5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

[48] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2008.10.027
[49] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114
[50] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514
[51] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2008, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453105
[52] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 521–530. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368160
[53] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510
[54] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Security*, 2015, pp. 17–26. [Online]. Available: http://dx.doi.org/10.1109/QRS.2015.14
[55] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Sci.*, vol. 313, no. 5786, pp. 504–507, 2006.
[56] H. K. Dam, T. Tran, J. Grundy, and A. Ghose, "DeepSoft: A vision for a deep model of software," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 944–947.
[57] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326
[58] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334
[59] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603
[60] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 1606–1612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060832.3060845
[61] 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

Hoa Khanh Dam received the bachelor of computer science degree from the University of Melbourne in Australia, and the master's and PhD degrees in computer sciences from RMIT University. He is a senior lecturer with the School of Computing and Information Technology, University of Wollongong (UOW), in Australia. He is associate director for the Decision System Lab at UOW, heading its Software Engineering Analytics research program. His research interests lie primarily in the intersection of software engineering, business process management and service-oriented computing, focusing on such areas as software engineering analytics, process analytics and service analytics. His research has won multiple Best Paper Awards (at WICSA, APCCM, and ASWEC) and ACM SIGSOFT Distinguished Paper Award (at MSR).

Truyen Tran received the bachelor of science degree from the University of Melbourne, in 2001, and the PhD degree in computer science from Curtin University, in 2008. He is associate professor at Deakin University where he leads a research team on deep learning and its applications to accelerating sciences, biomedicine and software analytics. He publishes regularly at top AI/ML/KDD venues such as CVPR, NIPS, UAI, AAAI, KDD and ICML. He has received multiple recognition, awards and prizes including Best Paper Runner Up at UAI (2009), Geelong Tech Award (2013), CRESP Best Paper of the Year (2014), Third Prize on Kaggle Galaxy-Zoo Challenge (2014), Title of Kaggle Master (2014), Best Student Papers Runner Up at PAKDD (2015) and ADMA (2016), and Distinguished Paper at ACM SIGSOFT (2015).

Trang Pham received the bachelor's degree in computer science from Vietnam National University in 2014. She is working toward the PhD degree at Deakin University. Currently, her research focuses on Recurrent Neural Networks for Structured Data.

Ng Shien Wee received the bachelor's and honours degrees in computer science from the University of Wollongong (UOW), in 2015 and 2016, respectively. He is currently working toward the PhD degree in the School of Computing and Information Technology and a member of the Decision Support Lab (DSL), University of Wollongong.

John Grundy is senior deputy dean of the Faculty of Information Technology at Monash University. His research interests include automated software engineering, software tools, human-centric software engineering, visual languages, software architecture, software security engineering and user interfaces. He is fellow of Automated Software Engineering and fellow of Engineers Australia. Contact him at john.grundy@monash.edu

Aditya Ghose received the bachelor of engineering degree in computer science and engineering from Jadavpur University, Kolkata, India, and the MSc and PhD degrees in computing science from the University of Alberta, Canada (he also spent parts of his PhD candidature at the Beckman Institute, University of Illinois at Urbana Champaign and the University of Tokyo). He is a professor of computer science at the University of Wollongong. He leads a team conducting research into knowledge representation, agent systems, services, business process management, software engineering and optimization and draws inspiration from the cross-fertilization of ideas from this spread of research areas. He works closely with some of the leading global IT firms. Ghose is president of the Service Science Society of Australia and served as vice-president of CORE (2010-2014), Australia's apex body for computing academics.