Code Analysis Using Deep Learning Techniques

***Abstract*—The static analysis of source code used to assist in many developer tasks. Many tools and approaches exist to perform this analysis, the majority of which rely on mappings from code elements and patterns to high level concepts. This mapping is typically a manual or semi-automated process which is usually quite costly and can easily fall behind the most recent source code versions and updates. We propose a technique to help further automate this process by modifying the skip-gram model to translate source code into word embeddings which can then be used as input to a deep learning model. We also describe how this technique can be evaluated to show its effectiveness.**

I. Introduction

The static analysis of source code is performed for many reasons, such as bug detection, security analysis, quality assurance, and more. Many static analysis tools and algorithms exist to assist in detection tasks on source code. However, the manual production of such tools and algorithms can be quite costly in time and money, and are prone to software bugs and human error. Although static analysis tools can be automatically applied to programs, they mostly rely on mappings from code elements to high level concepts to detect bugs related to software semantics. For example, Flowdroid [1] relies on SuSi [2] to provide the mapping from source and sink API methods to information types, or *pscout* [3] to provide the mapping from API methods to permissions.

In existing techniques, the mapping is typically generated manually or semi-automatically (e.g., SuSi is automatically extended from a manually built training set). Besides the huge manual effort required for constructing such mappings, they also have the following three limitations. First of all, libraries and frameworks are often evolving frequently, so the mappings can go out-of-date quickly. *pscout*'s developers have maintained their mappings since 2012, but its most updated mapping still only supports up to Android 5.1.1, which was released in April 2015 (more than three years from now). Similarly, SuSi only has a version for Android 4.2, which was released in July 2012 (more than six years from now). Second, construction of such mappings is doable only when there exists a common library or framework for certain tasks. For example, Android SDK is just about the only way to collect information from an Android device (with native code as exceptions). By contrast, a lot of areas have multiple or many libraries (e.g., encryption and authentication libraries, J2EE implementations), making it difficult to pre-construct a mapping for them. Third, such mappings can only handle API methods whose high-level semantics are known before the analysis. However, in many scenarios code analysis needs to know high-level semantics of methods or variables defined by client software developers. For example, a code security checking tool may need to know which variables represent user name and password so that it can check whether they are well protected (e.g., encrypted when stored locally and transmitted through secure connection protocols).

Therefore, a technique that can construct mapping from code elements to high-level concepts without prior preparation will alleviate the above limitations and significantly enhance usability of code analysis tools in practice. In this paper, we describe a technique to map arbitrary code elements to a set of pre-defined high-level concepts. As a basic solution, we can always measure textual similarity between keyword / phrases representing the concept and the identifiers in the code, and for each keyword k we can retrieve all code elements having a similarity with k above a certain threshold. However, developers may use different identifiers to represent the same concept or even use identifiers that are not meaningful (when good coding style is not enforced). So a simple retrieval technique will have rather low recall.

The novel intuition behind our approach is that, given a high-level concept, similar to its textual context (i.e., texts surrounding a keywords / phrase representing the concept), its code context (i.e., code elements surrounding a variable / method representing the concept) also contains a relationship between it and its relevant concepts. For example, an assignment $a = b$ indicates that the concept represented by b is a sub-concept of the concept represented by a, and an assignment $a = b[i]$ indicates that the concept represented by b is a collection, and its element is a sub-concept of a. With all these relations from context, we can gradually narrow down the possible concepts a code element can represent by comparing a concept's textual context with its code context. In our approach, we use Word2Vec [4] and deep learning to represent the code / textual context of a concept, as described in detail in Section III.

Motivation Example. Figure 1 presents an example for computation and clustering features. A user input is first checked with PhoneNumberUtils.isGlobalPhoneNumber(), and then processed by PhoneNumberUtils.parse().getCountryCode(). Both API methods are mainly used for processing phone numbers, so with them as computer features, we can predict the user input as a phone number even if it does not have a meaningful identifier. Later, this phone number flows to a field called pNumber in class Profile. From the other fields defined in the class (e.g., address) also indicates that this field may be a phone number as they are all components of contact information. From the defined fields, we can also infer that the field zCode should be zip code.

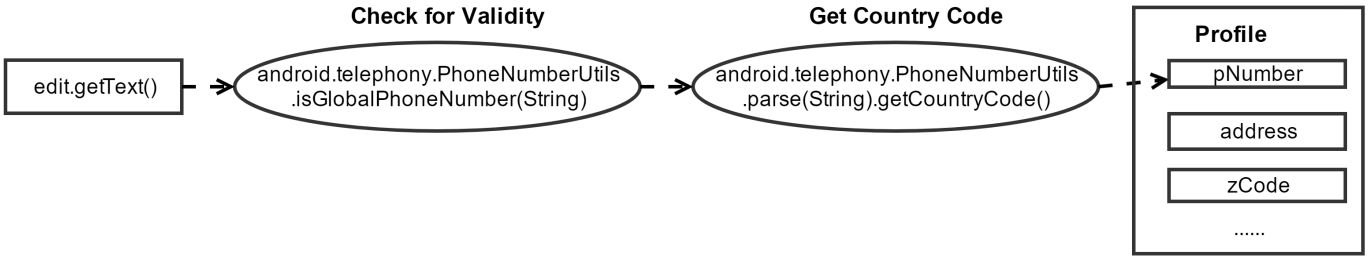

Fig. 1. An Example of Code Context

II. Background

This section contains background information necessary on which our approach is based.

A. Word Embeddings

In natural language processing, word embeddings are used to represent the semantic meaning of word. A word embedding is an n-dimensional vector that gives a distributed representation of a word with each vector dimension representing some semantic feature of the word. Words with similar semantic meaning will be clustered together in the vector space since similar words will be used in similar context [5].

GloVe (Gloval Vectors for word representation) [6] and Skip-gram [4] are the two most popular word embedding learning models. Both operate on the same fundamental idea that the basic semantic meaning of a word can be represented by utilizing surrounding words as semantic context. GloVe uses a co-occurance matrix of all the words in a corpus. When the matrix has been filled with all word co-occurances, matrix factorization is performed to produce the vector values of each word's embedding. Skip-gram defines a window size that indicates the number of surrounding words to use as context. A given word's vector values are updated by using its context as input to a neural network.

After a corpus has been input to either of these algorithms and all words have been processed, the resulting word embeddings should have grouped similar words together in the vector space. That is, the closer two words are to each other, the more similar their semantic meaning is. For our purposes, we will use the popular skip-gram algorithm Word2Vec. We propose to use a modified version of Word2Vec to obtain word embeddings for words in source code.

B. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a neural network that processes sequences of information in steps [7]. For each step in an RNN, new input is processed with the input that came before it. That is, for step t the new input x_t is passed to the hidden state, h_t, of step t. h_t then uses x_t and the output of the previous step's hidden state h_{t-1} to calculate the current output using the function $Vx_t + Wh_{t-1}$. The result is then usually transformed using a nonlinear function f, such as $tanh$. So, the final calculation for the hidden layer (and the output of that layer r_t) is $r_t = h_t = f(Vx_t + Wo_{t-1})$. The hidden state of the current step is then used in the next step's calculation with the next input, x_{t+1}. The steps continue until all input has been consumed and the final output is usually transformed by a softmax classifier to determine the classification (or final analysis) of the input.

III. Approach

In this section we discribe a modified skip-gram model that would better facilitate the embedding of words from source code than the normal skip-gram algorithm. We also discuss some preprocessing steps of the source code before using it as input. For the sake of simplicity and readability we will use Java as the example programming language during discussion for the rest of the paper. However, it should be noted that these ideas can be applied to any programming language with some modification.

A. Preprocessing

The source code will need to be preprocessed before being used in the skip-gram model. The main preprocessing step is the identification of the scope of variables and functions. For example, given a particular class in Java, a class variable can have the same name as a local variable. Similarly, two different classes may have the same function name. This is not a problem for natural language as every word in a natural language is unique. While having variables and functions with the same name is legal in Java (within certain rules), it can heavily skew the values of the word embeddings as all variables and functions with the same name will be mapped to the same word embedding. To account for this problem, we propose to extend all variable and function names with the class and function they belong to based on their scope. For example, given the Java class *String*, its function *charAt* will be extended to *String.charAt*, and the local variable *index* in the function *charAt* will be extended to *String.charAt.index*. This will ensure that all variable and function names are mapped to their own word embeddings.

However, this causes a new problem. If variable and function names are extended to be specific to the class they are defined in, then when analyzing new software there will be no word embeddings for any newly defined variables and functions. To remedy this, new software must first be input through the skip-gram model to produce the new necessary word embeddings before being input to a deep learning model.

Other preprocessing includes the removal of all unnecessary white space, some punctuation, and comments. We aknowledge that there are other possible preprocessing steps that can be taken, but we do not consider these at this time. For example, in Java it is legal to overload functions (that is, have different definitions and parameter lists for the same function name). We consider overloading the same as a word in natural language having multiple definitions. While the function is somewhat disambiguated, we believe it will not disambiguate enough to drastically change the results of a deep learning analysis.

B. Modified Word2Vec

As described earlier, Word2Vec uses a window size to determine the context of a word. This works fairly well for natural language processing because each word has a general predefined meaning and the syntax of a sentence does not allow for complex syntactic structures such as decision or loop statements. So a window size that only scans for context a bit before and after a word is able to determine a relative semantic meaning to other words. However, in source code variables and functions are defined within their class (and not always before they are used in the case of functions) and there are more complex syntactic structures. This seems to imply that a simple window size that scans before and after a word will not be sufficient in determining proper word embeddings for source code.

To modify skip-gram, we will need to separate "key words" (which includes key words, punctuation, and operands) from "constructed words" (which includes class, function, and variable names) defined in libraries or by programers. Key words are usually handled by the compiler and are not explicitly defined in a library, unlike the constructed words discussed later. We further divide key words into two categories, those that have a syntactic structure associated with them and those that do not. Such key words that have syntactic structures are *if-else*, *while*, *for*, etc. These key words derive most of their meaning from their syntactic structure. Therefore, instead of using a window size, Word2Vec will consider everything that is a part of the structure as context when updating the word embedding values of that key word. For example, in the statements *for(i; i < 10; i++){ System.out.println(i); }* everything between the parenthesis after the *for* and before the { will be considered as context for the key word *for*. It is important to note that we do not consider the statement within the loop structure (*System.out.println(i);*) as part of the context for the key word *for*, as the statements within key word syntactic structures do not usually have a direct impact on the meaning of the structure (and the key word associated with that structure). For the key words that do not have a syntactic structure (*break*, *+*, *&&*, etc.), only context before and after the word is available to derive meaning. Therefore, Word2Vec will be used normally for these key words.

A constructed word's embedding will be use context in two separate ways. The first, and most important, is that word's definition. When a constructed word is defined, the entire definition is used as context to the skip-gram model. For example, for a Java class everything in the entire class is used as context; for a function, the function signature and entire definition will be used as context; for a variable, any assignment performed on it will be used as context. The second type of context will come from the constructed word's usage in code. In this case, Word2Vec will be used normally with a window size to capture code surrounding the word as context.

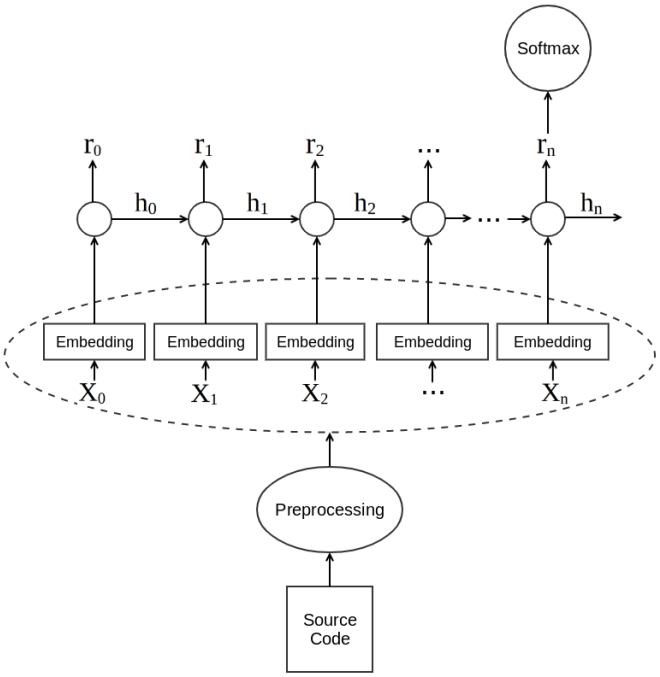

Fig. 2. Flow of Deep Learning Analysis

C. Deep Learning

Once preprocessing and word embeddings have been completed, the source code is ready to be analyzed by a neural network. The network we will use is an RNN. This is appropriate since the current state of code at any time is dependent on what statements have been executed before it. Starting in the main function, for each word in the code, the embedding associated with that word will be used as input to the next step in the RNN. In the next section we will discuss how we plan to evaluate the effectiveness of deep learning on source code using the described approach.

IV. Evaluation

The substantial number of Android apps in the Google Play store and open source software repositories provide us with a large dataset for mining of identifier similarities, training classification model, and evaluation. The major challenge is to obtain ground truth of privacy-related code elements for both training and evaluation purposes. As mentioned earlier, we plan to use most explicit privacy-related code elements for training purposes. For evaluation, besides manual confirmation of the results, we plan to further execute the app and track the

flow of privacy data (both from user inputs and Android APIs) with taints, so that we can confirm whether a variable holds privacy data or a method processes privacy data at runtime.

It is also important to evaluate whether our mappings are useful for developers. We will implement our approach as a plug-in of Android Studio plugin which recommends privacy-related facts to developers. With training materials on how to handle each type of privacy code entities, we can further help developers find potential bad coding practices regarding privacy. We plan to evaluate the usability of the plugin with human developer subjects, and apply our plugin to open source Android projects and report potential privacy-related coding issues to developers.

V. Conclusion

We have described a novel approach to utilize deep learning techniques to assist in the analysis of source code. A process for translating words in source code to word embeddings to be usable as input has been detailed. Evaluations to determine the effectiveness and feasability of the techniques have also been explained.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[2] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks." in *NDSS*, 2014.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[5] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[6] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[7] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.