Understanding the Meaning of Code Elements with Deep Learning

Abstract—**Program analysis on software source code can be used to assist in many developer tasks. Many tools and approaches exist to perform this analysis, a large portion 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 library API 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 First, libraries and frameworks are often evolving frequently, so the mappings can go out-of-date quickly.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 ago). Similarly, SuSi only has a version for Android 4.2, which was released in July 2012 (more than six years ago). Second, construction of such mappings is doable only when there exists a common library or framework for certain tasks. For example, Android SDK is about the only way to collect information from an Android device (with native code as exception). By contrast, a lot of areas have many libraries (e.g., encryption and authentication libraries, J2EE implementations, etc.), 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 mappings 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 keywords or 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 keywords or phrases representing the concept), its code context (i.e., code elements surrounding a variable or 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 context of a concept, as described in detail in Section III.

Motivating Example. Figure 1 presents an example for code context. 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, and they have the keywords in their names. So with them as context 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. The other fields defined in the class (e.g., address) also indicate that this field may be a phone number as they are all components of contact information. From the defined fields we can also infer

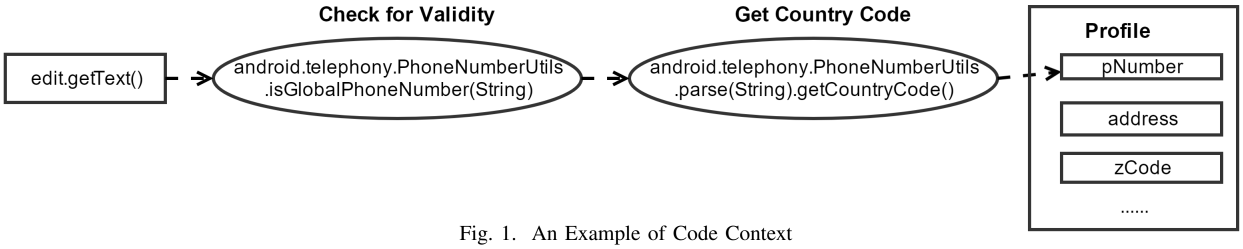
Check for Validity
Get Country Code
edit.getText()
android.telephony.PhoneNumberUtils
.isGlobalPhoneNumber(String)
android.telephony.PhoneNumberUtils
.parse(String).getCountryCode()
Profile
pNumber
address
zCode
......

Fig. 1. An Example of Code Context

that the field zCode should be zip code.

II. Background

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

A. Word Embeddings

In natural language processing, word embeddings are used to represent the semantic meaning of words. A word embedding is an m-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 two of the 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. 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 neural networks that process sequences of information in steps [7], shown in the RNN section in Figure 2. 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 step, r_t) is $r_t = h_t = f(Vx_t + Wh_{t-1})$. 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, at which time the final output is produced. The final output can be used in a number of ways. For example, for a classification problem the output is usually transformed by a softmax classifier to determine the classification of the input.

C. Related Work

To our knowledge, we are the first to use deep learning techniques to analyze source code. There have been many applications of deep learning techniques used in natural language processing [8] [9]. The popularity of applying these techniques to analyze natural language has grown rapidly in recent years. Before the use of deep learning, the main focus of analysis of natural language was the use of statistical language models. Hindle et al [10] showed that the same statistical language models used to analyze natural language were just as useful (if not more so) in analyzing code in software. Since analysis of natural language has made a shift to using deep learning techniques from statistical language modeling, it is worth exploring the application of deep learning on software.

Statistical machine translation (SMT) is based in statical language models and has been widely used to analyze and translate source code. Nguyen et al [11] conducted experiments that attempted to translate Java code to C# code using SMT. Source code was treated as a simple sequence of tokens and was translated on a method-by-method basis. The results showed a need for improvement, but indicated potential. More recently, Vasilescu et al [12] described a tool, *Autonym*, which used SMT to recover orignal names from minified names in JavaScript programs. The results were comperable to other state of the art deobfuscation tools, however the names *Autonym* obtained easily were difficult for other tools to obtain. Conversly, the names *Autonym* had trouble obtaining were easily identified by other tools. This lead the authors to blend their tool with another, *JSnice* [13], which lead to a significant improvement in results.

III. Approach

In this section we discribe the general flow that a piece of software would go through in our proposed setup, shown by Figure 2. 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 as input to Word2Vec. Preprocessing is represented by the brown section in Figure 2. The initial preprocessing step is basic text cleaning. This includes the removal of all unnecessary white space, comments, and some punctuation. Most punctuation is kept as much of it is necessary in determining the proper structure of the source code. Some of it will also

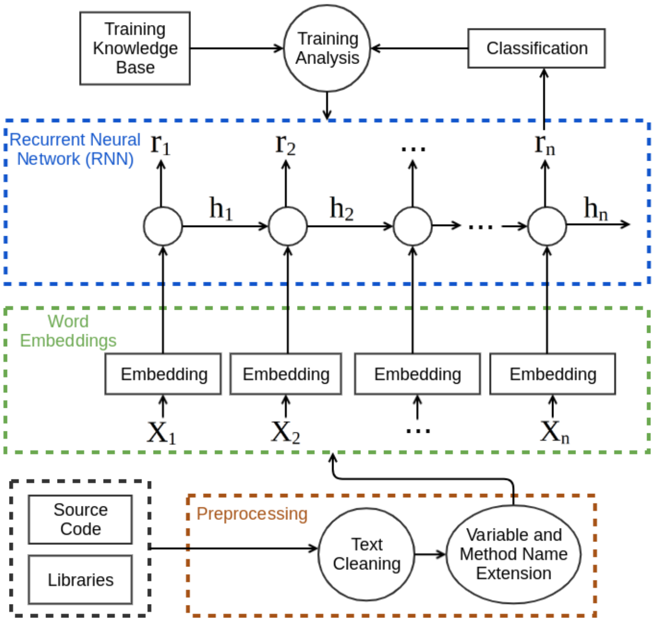

Fig. 2. Flow of Deep Learning Analysis

be necessary for the modified Word2Vec algorithm to perform properly.

The main preprocessing step is the identification of the scope of variables and methods. 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 method name. This is not a problem for natural language as every word in a natural language is unique. While having variables and methods 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 methods with the same name will be mapped to the same word embedding. To account for this problem, we propose to extend all variable and method names with the class and method they belong to based on their scope. For example, given the Java class *String*, its method *charAt* will be extended to *String.charAt*, and the local variable *index* in the method *charAt* will be extended to *String.charAt.index*. This will ensure that all variable and method names are mapped to their own word embeddings. However, this causes a new problem. If variable and method 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 methods. To remedy this, new software must first be input through Word2Vec to produce the new necessary word embeddings before being used as input to a deep learning model.

We acknowledge 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 methods (that is, have different definitions and parameter lists for the same method name). We consider overloading the same as a word in natural language having multiple definitions. While the method is somewhat ambiguated, we believe it will not ambiguate 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 methods are defined within their class (and not always before they are used in the case of methods) 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.

Word embeddings are represented by the green section in Figure 2. To modify Word2Vec to process source code more effectively, we will need to separate "keywords" (which includes keywords, punctuation, and operands) from "constructed words" (which includes class, method, and variable names) as keywords are usually handled by the compiler and are not explicitly defined in a library, unlike constructed words. We further divide keywords into two categories, those that have a syntactic structure associated with them and those that do not. Such keywords that have syntactic structures are *if-else*, *while*, *for*, etc. These keywords 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 keyword. For example, in the snipet *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 keyword *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 keyword *for*, as the statements within keyword syntactic structures do not usually have a direct relation on the meaning of the structure (and the keyword associated with that structure). For the keywords 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 with a window size for these keywords.

A constructed word's embedding will 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 for Word2Vec. For example, for a Java class everything in the entire class is used as context; for a method, the method 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.

C. Deep Learning

Once preprocessing and word embeddings have been completed, the source code is ready to be analyzed by a neural network, represented in the blue section in Figure 2. 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 method, for each word in the code, the embedding associated with that word will be used as input to the next step in the RNN.

After all words in the source code have been processed, the final output can be used to determine whatever the RNN is attempting to learn. In Figure 2, a general example for the final steps to determine mappings is shown. The final output from the RNN is transformed using softmax to classify the mapping that is most likely associated with the input code. During training, the classification determined by the RNN is analyzed against the ground truth, represented by the training knowledge base. For any incorrect classifications made by the RNN, a loss is determined which is then used to update the RNN weights through backpropagation. This should result in better classifications in the future.

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 the training of a classification model and the evaluation of our approach. To evaluate our approach, we need to apply it on a concrete application scenario, so that we have a known set of high-level concepts and keywords. We plan to evaluate our approach first on the mapping of information types to code elements in Android security and privacy analysis. Existing works [2] [14] have enumerated a number of high-level concepts and keywords for information types that Android apps collect.

The major challenge is to obtain ground truth of privacy-related code elements for both training and evaluation purposes. In particular, as mentioned earlier, we plan to use most explicit code elements for training purposes. For evaluation, we plan to manually check the retrieved code elements for each high-level concept. 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.

V. Conclusion and Future Work

We have described a novel approach to utilize deep learning techniques to assist in the analysis of source code by translating words in source code to word embeddings, using those embeddings as input to a deep learning model, and evaluating the effectiveness and feasability of the approach on privacy detection. Besides providing semantics support to static analysis tools, our approach may enable and lead to a large variety of research projects and techniques in software engineering. First, the mapping from code elements to high-level concepts can help developers better understand their code by automatically adding comments and commit messages or by retrieving relevant code in code search. Second, such automatically generated mappings may facilitate the reuse of code and patterns across project boundaries, as they provide an intermediate point (i.e., textual keywords representing high-level concepts) to map code elements from different projects. Third, nowadays software is used more and more commonly in various embedded environments (e.g., smart cars, medical devices, etc.) These environments need to follow privacy and safety regulations specific to those areas. The mappings are able to link high-level concepts in privacy and safty regulations to concrete code, and thus enable automatic enforcement of regulations for embedded software.

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.

[8] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[9] X. Glorot, A. Bordes, and Y. Bengio, "Domain adaptation for large-scale sentiment classification: A deep learning approach," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 513–520.

[10] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[11] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 651–654.

[12] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 683–693.

[13] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[14] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "Supor: Precise and scalable sensitive user input detection for android apps." in *USENIX Security Symposium*, 2015, pp. 977–992.