Toward Detection of Access Control Models from Source Code via Word Embedding

John Heaps
The University of Texas at San Antonio
john.heaps@utsa.edu

Xiaoyin Wang
The University of Texas at San Antonio
xiaoyin.wang@utsa.edu

Travis Breaux
Carnegie Mellon University
breaux@cs.cmu.edu

Jianwei Niu
The University of Texas at San Antonio
jianwei.niu@utsa.edu

ABSTRACT

Advancement in machine learning techniques in recent years has led to deep learning applications on source code. While there is little research available on the subject, the work that has been done shows great potential. We believe deep learning can be leveraged to obtain new insight into automated access control policy verification. In this paper, we describe our first step in applying learning techniques to access control, which consists of developing word embeddings to bootstrap learning tasks. We also discuss the future work on identifying access control enforcement code and checking access control policy violations, which can be enabled by word embeddings.

ACM Reference Format:
John Heaps, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. 2019. Toward Detection of Access Control Models from Source Code via Word Embedding. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3322431.3326329

1 INTRODUCTION

The ability to verify access control policies and properties of a system is invaluable in assuring the security and proper compliance of sensitive data access and system performance. While manual verification is always an option, it is prone to human error and requires large amounts of time for even average-sized systems, making such a process infeasible for many systems. Thus, automation of this verification process has been a topic attracting much research. Many current techniques rely on static or dynamic analysis of a system to identify and build access control models that define the system's access control behavior. By creating an access control model, verification of the access control policy becomes a much simpler task. However, these analysis techniques suffer from many obstacles in building such models, such as determining what code elements are related to roles or permissions and how to link these code elements to specific policy elements. Further, even if these mappings are defined for a system, static analysis techniques will only be able to detect those links that are very similar in implementation. That is, static analysis does not have the ability to determine or construct new mappings, only those that are manually defined.

There have been advancements in machine learning, specifically deep learning, in recent years. This includes novel applications in learning classification and prediction tasks dealing with image processing, text and sentiment identification, network traffic analysis, and others. While still very new, research in deep learning on source code is becoming increasingly popular. This includes preliminary work done on clone detection [15], function naming and summerization [3], vulnerability detection [13], and more. We believe that the techniques and algorithms being developed have potential to assist in automating access control policy verification. For example, we hope deep learning is not only able to learn the patterns necessary to determine links between code and policy elements from annotated training data, but that it can also find new links between code and policy elements previously unencountered.

Like natural language text, code elements do not contain underlying numerical meaning for the necessary deep learning calculations to be performed. Therefore, in order to perform deep learning tasks on access control, the first step is to create word embeddings for code elements. Word embeddings map code elements into a high dimensional space to facilitate learning. In this paper, we describe our initial attempts to construct word embeddings using a modified version of the popular Word2Vec algorithm. Word2Vec creates an embedding given a word and its surrounding words, called the word context. Our modifications primarily focus on how to construct word context for code elements so that high quality word embeddings are produced. We run the modified Word2Vec algorithm on the Java Development Kit 8 (JDK8)[1] and the access control library Apache Shiro[2], and evaluate the resulting embeddings by reporting the perplexity and showing a plot of 99 of the most common code elements in each library.

We also discuss future work such as how we plan to further improve our word embeddings for code elements which mainly focus on modifications to the Word2Vec model in addition to changes to the input. Furthermore, we present our plan to perform evaluations on the capabilities of the word embeddings and how the embeddings can be applied to mapping access control policies to code implementation and checking for policy violations. Finally, a

SACMAT '19, June 3–6, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6753-0/19/06…$15.00
https://doi.org/10.1145/3322431.3326329

[1] https://www.oracle.com/technetwork/java/javase/overview/index.html
[2] https://shiro.apache.org/

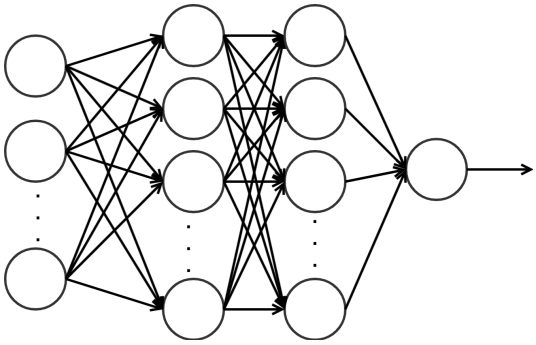

Figure 1: Neural Network Example

list of major problems that will be encountered with deep learning on source code is also identified and discussed.

2 BACKGROUND

In this section, we present background on deep learning, word embedings, and discuss related work.

2.1 Deep Learning

A *neural network* is a feed-forward graph consisting of multiple layers, where each layer is made up of a number of neurons. There are three main types of layers, each with corresponding neurons: input, hidden, and output. The input layer is composed of input neurons that define the shape of the input data to the network. The hidden layers are composed of neurons that perform a transformation on the input data that can be used to determine the final output of the network. Each hidden layer has an activation function associated with it and each neuron in the layer has a weight and bias associated with it. The weight and bias are used to calculate a linear transformation on the input which is then used as input to the layer's activation function, such as the `sigmoid` or `tanh` functions. The output layer is composed of output neurons which interprets the final results of the hidden layer transformations and produces the final classification or prediction of the network. Figure 1 shows an example of a neural network. Each vertical line of nodes is a layer. The leftmost layer is the input layer, the middle two layers are hidden layers, and the rightmost layer is the output layer.

After the architecture of the network is determined, a network is trained by defining the set of inputs and corresponding proper outputs, called labels. Next, an input is given to the input layer of the network and the output layer gives a classification or prediction. The network's output is then compared to the corresponding input label. If the answer is incorrect, a loss function determines how far from the label the network's output was, and the weights and bias are modified based on the calculated loss using back prorogation. This learning happens over multiple iterations of the entire set of input. Ideally, the weights will be modified so that for each input the network's output constantly matches that input's label.

Neural networks perform well when learning over data that is easily represented with numerical values, such as: individual pixels in a picture, network traffic data, metric data, etc. For text, however, there is no meaningful underlying numerical values to represent words. To solve this problem, natural language processing utilizes word embeddings to represent the relative semantic meaning of words. A *word embedding* is an m-dimensional vector of real numbers, with each dimension representing some semantic feature of words which gives a distributed representation of a word [10]. Each vector specifies a word's location in the m-dimension space, and words with similar meanings are close in this space. Since we use deep learning to determine the values for each word's vector, we currently have no way to determine what feature each dimension represents. What we are able to determine, however, is whether those word embeddings have been assigned relatively meaningful values based on their location in the space. For example, take the words *king* and *queen*. These should be in similar locations in the vector space as they both describe a ruler of a monarch, only separated by the distinction of the king being male and the queen being female. Similarly, the words *man* and *woman* should be in fairly close proximity to each other as they both describe a human being, again separated by the man being male and the woman being female. When looking at meaningful word embeddings, not only will we find *king* and *queen* close together and *man* and *woman* close together, but we should also expect that the distance between *king* and *queen* should be about the same as the distance between *man* and *woman*, since they are separated by the same concept of gender. That is, if our word embeddings were perfect, then the statement $queen = king - man + woman$ would hold, using the word embedding vectors that represent each word. Word embeddings should describe the relative meaning between the words in the vocabulary. This will allow learning models to find the relations and patterns in textual data that achieve optimal outputs for assigned learning tasks. These relative meanings between all the words in the vocabulary are realizable because there is a naturalness and structure to language. As described by Hindle et. al. [7], this same naturalness and structure are mimicked in programming languages as well, except there is even greater naturalness and stricter structure in programming languages than in natural languages. This suggests that word embeddings should also be applicable to source code as well as normal text.

GloVe (Global Vectors for word representation) [12] and Skip-gram [9] are two of the most popular word embedding learning models. Both operate on the same fundamental idea that, given a corpus of text, the semantic meaning of a word can be determined by utilizing the surrounding words as semantic context. GloVe uses a co-occurrence matrix of all the words in a corpus. When the matrix has been filled with all word co-occurrences, 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 predicting its context using a neural network. For our purposes, we will use the popular Skip-gram algorithm Word2Vec, and propose to use a modified version of Word2Vec to obtain word embeddings for code elements in source code.

2.2 Related Work

In our related work, we separately discuss the applications of deep learning on source code by the way word embeddings have been constructed. We first describe work where word embeddings were

learned using an unmodified version of the Word2Vec or Skip-gram algorithms, and then discuss work where the algorithms were modified to more appropriately handle the structure of source code.

2.2.1 Deep Learning On Source Code. Below, we review work wherein word embeddings were directly learned from the source code without using specialized data structures. The embeddings were learned linearly. That is, each code element used surrounding code elements from in front and behind it to determine its embedding. This is a very different approach from ours, which learns embeddings based on a set of rules for each type of code element, and which has important issues that we discuss in Section 3 and Section 5.1.

Hellendoorn et al. [6] examined the feasibility of using deep learning on source code. They compare deep learning models with statistical models and mixed models (i.e., a combination of statistical and deep learning models) over performance on source code word prediction. They show that deep learning has no major advantage over statistical models, but mixed models improved accuracy to outperform other model types. They identify a number of issues that deep learning has with source code and address some of the limitations. The highest accuracy achieved by the code prediction experiments was 86.2% by the mixed models. However, the learning used basic n-gram (i.e., skip-gram) which only uses the nearby, surrounding code elements to determine the word embeddings used to perform code prediction. This ignores the structure of code which could have been utilized by the learning algorithm to yield better results.

White et al. [16] learns a software language model (i.e., word embeddings) using a recurrent neural network (RNN) (without long-short term memory (LSTM)). They discuss motivations for using deep learning and the construction of their RNN model. The RNN is compared to a "state-of-practice" n-gram algorithm which the RNN outperforms as measured by perplexity and in a code suggestion experiment. The highest accuracy achieved during code suggestion was 92% for top-10 suggestions and 88.4% for top-5. While this approach shows potential, it would be interesting to see it perform on tasks other than code suggestion or prediction. Using an RNN to learn word embeddings linearly is likely a good option for code suggestion, but for many other tasks this approach may not perform as well.

Russell et al. [13] performs a vulnerability detection task on C/C++ code. They create two datasets using a combination of static analysis and manual inspection to label code functions as *vulnerable* (meaning the funtion contains a bug pattern that could cause security risk) or *not vulnerable* (meaning the function does not contain any bug pattern that could cause a security risk). One dataset is constructed using a combination of Debian Linux distribution and GitHub project source code, and the other dataset is constructed using the SATE IV Juliet Test Suite[3]. After building this dataset, they build word embeddings by parsing C/C++ source code into a vocabulary of code elements and then reduce that vocabulary to 156 representative tokens, which then have word embeddings constructed for them. They then use these word embeddings in a convolution neural network (CNN) classifier to learn and predict if a given function is *vulnerable* or *not vulnerable*.

[3]https://samate.nist.gov/SATE4.html

The major difference between these models and ours is the learning of word embeddings using context of immediately surrounding code elements. In our approach we utilize an abstract syntax tree (AST) which allows us to take advantage of the syntactic structural information of source code.

2.2.2 Deep Learning Models Utilizing Abstract Syntax Trees For Code Analysis. Peng et al. [11] describe building vector representations (i.e., word embeddings) for program code. These vectors can be used as input to a deep learning model in order to assist the model in its learning and analysis. They build such vectors using a "coding criterion" model on nodes in an AST representation of the program code. The coding criterion dictates that a non-leaf node in the AST is a function of the sum of all its children nodes, weighted by the depth of the child from the parent (i.e, the further away from the parent in the AST, the less weight it receives). The identifiers are not used (e.g., names, types, etc.), only the AST node types (e.g., FuncDef, BinaryOP, Decl, etc.). To test their word embeddings, they performed a classification task on code using the vector representations in a CNN. Programs were labeled with an ID and the CNN was tasked with learning which programs were assigned to which IDs. The CNN performed at 95.33% accuracy. While the results show promise, there were only four programs in this experiment, which did not define the program size. While our approach utilizes node name and type, this approach only used the AST node types throughout the learning process. It is unclear how this might perform in other larger and more complex learning tasks.

White et al. [15] performs clone detection using deep learning. They model the lexical code features and structural code features separately using RNN and recursive neural networks, respectively. The lexical features of the code are processed in linear order to define a language model. The structural features of the code are processed using an AST, which is converted to a binary tree and then annotated with the learned language model. They performed the clone detection on eight Github Java projects, six of which had a precision above 90% and the remaining two had precision scores just below 60%. While these results show promise, the only metric described in the paper was precision, leaving out other scores normally reported, such as accuracy, recall, F1, etc. The learning of lexical features using n-gram (which only learns linear context) and then annotating structural features (non-linear context) with these may also hinder the learning algorithm. In our approach, all context is learned utilizing an AST as opposed to learning some context linearly and other context with an AST.

Alon et al. [3] perform function naming using paths of the function declaration and body. They encode each path as an embedding which is then learned over all given functions. While this approach yields good results for most function naming, there were some names that were as low as 40% probability that the correct name was chosen. Further, it is not clear how this would scale to lower level or higher level code elements (i.e., how would it determine paths for a class or for a single variable that can appear in many levels of an AST). Our approach learns individual code elements instead of code paths, which allows better scalability.

Finally, Tai et al. [14] describe a tree LSTM neural network. The LSTM nodes are able to receive input from multiple child nodes that

utilize forget gates. The approach is very interesting, but the learned embeddings are based purely on the child nodes. In Section 3, we discuss problems with this type of model. However, the techniques and strategies may help in producing better word embeddings for particular AST node types.

3 CURRENT APPROACH

In our current work, we created word embeddings for Java code elements using Tensorflow [1]. These word embeddings were created using a modified version of the Word2Vec skip-gram algorithm. In original skip-gram, a dictionary is defined which is an indexed list of all words in a corpus. These indices correspond to a list of vectors that are the word embeddings. The word embeddings are initialized to random values which will be modified by the Word2Vec algorithm using a neural network. For each word in the corpus, context is defined that determines the word's semantic meaning. Given a target word in the corpus, the context for that word is found by a window that looks a number of words before and after the target word. In original skip-gram, each context word is used as a label for the target word. The neural network receives the target word as input and tries to predict each label as output. The word embeddings are then modified during back prorogation.

Since a dictionary can be quite large, it can take a large amount of time for each prediction and back propagation task to be performed. This makes learning infeasible for any average-sized corpus. Word2Vec solves this problem by utilizing negative samples. That is, Word2Vec pairs each context word with the target word to be used as *related* pairs, but it then also chooses random words from the dictionary that are not present in the context to pair with the target word to act as *unrelated* pairs. Instead of providing the target word as input and predicting the contex as output, Word2Vec provides the target word and either a context word or random word as a pair as input and predicts whether the pair is *related* (the second word was a context word) or *unrelated* (the second word was a random word). This changes the original prediction task from distinguishing between every word in the entire dictionary to a binary determination. This drastically reduces the time to perform prediction and back propagation and allows the algorithm to learn embeddings quickly for a large corpus.

For our purposes, the Word2Vec algorithm is used as described with the exception of how the context for each code element is chosen. In natural language, using surrounding words as context for a target word is motivated by the idea that words with similar semantic meaning will be used in similar ways. This implies that words with similar semantic meaning will be generally surrounded by the same or similar words, which will then define the same or similar context for the prediction task. However, in many cases this is not true for code. The complex syntactic nature of code causes code elements that should be used as context for a target code element to appear in many different places that are not in the immediate vicinity of the target code element. For example, in Java if only the surrounding code elements were used as context for a method name, only the method's modifiers and parameters would be captured and used as context; however, the entire method definition should also be used as context as those code elements directly determine the meaning of the method name. Similarly, certain code elements should not use some surrounding code elements as context. For example, the statements before and in the body of a while loop should not be considered as context for the *while* keyword since the statements do not affect the behavior of *while* (with the exception of the *continue* and *break* keywords). Therefore, a different strategy for choosing the context for each code element in the network must be utilized.

The syntactic structure of code is quite complex compared to natural language, and is key in determining how code elements are related to each other, as shown by Peng et al. [11]. Therefore, to help capture the inherent structure of code, we determine code element context using a code's AST. An AST represents source code as a parse tree based on the context-free grammar of the language. We use the JavaParser[4] tool to obtain the AST of a Java program.

In our first attempt at creating word embeddings for code elements, we assumed that for a given node (or code element) in the AST, all of its child nodes would provide the context for that node, similar to the work done by Peng et al. [11] and Tai et al. [14]. However, the word embeddings generated by this approach were of poor quality. We identified a number of issues with this approach, and two specific issues stand out. The first issue is that certain code elements will always be leaf nodes (such as *break* or labels) and therefore will never have any context for the neural network to learn. The second issue is that certain children do not actually affect the meaning of their parent, such as the example given earlier for the while loop, wherein statements inside the while loop (which are children of the while node in the AST) do not determine the meaning of the *while* keyword. We also determined that prior work that attempts to generalize the process of choosing context for all types of nodes had their own issues. We have observed that a single strategy for choosing context for all types of nodes in the AST will ultimately lead to certain node types pairing their associated code elements with incorrect context or missing important context that should be paired with those node types. This significantly degrades the quality of the word embeddings, which will negatively impact any learning task upon which the embeddings are used.

We propose instead to use sets of rules to determine the context for code elements based on their node types. For example, JavaParser has 83 node types to represent the grammar of Java. Types of nodes include, but are not limited to, nodes for: comments, literals, loop statements, decisions statements, class declarations, method declarations, variable declarations, and equation operators. We constructed rules for node types by asking the question, "What other node types or code elements are needed to determine the meaning or perform the task of this node type?" For example, a method declaration node needs its modifiers, parameters, and method body to determine its meaning; a while statement node needs its conditional expression and any statements that are present that directly affect its behavior (such as a *continue* or *break* statement) to perform the loop process properly; and a binary operator, such as *plus*, needs the expression of each of its terms. We have included Table 1 that details what code elements were included as context for each node type. In the table, columns 1-3 present the type of AST node, identifier of the AST node, and code elements being used as the

[4] http://javaparser.org/

context of the AST node. In our current approach we did not include embeddings for annotations, comments, imports, packages, lambda expressions, generics, or variable names.

```
//check if the user is an admin
public boolean isAdmin(){
     if(role == Role.Admin){
          return true;
     }
     else{
          return false;
     }
}
```

Figure 2: Example Code Snippet of a Function *isAdmin*

Figure 2 presents the code snippet for a function called *isAdmin*. The AST representing the code snippet is shown in Figure 3. This function is part of a *Subject* class that defines a user of a system. The class has a *role* field that defines the role of the user. This function is simply checking whether the user has the role *ADMIN*. In our algorithm, we walk the tree, visiting each node. As we visit a node, we call the current node we are visiting the *target code element* and we next extract *context code elements* that help define the semantic meaning or behavior of the target code element. In this example, the current target code element is an If Statement. If we look at Table 1, the identifier in the dictionary for an If Statement is if and the context is the conditional expression and the *else* identifier if it is present. The condition expression is a Binary Expression, so we record as context the operator and the left and right terms. As stated above, we currently do not include variable names in our embeddings, instead we record the variable data type. Therefore, the context that will be recorded for the if node will be *EQUALS* (the binary operator), *Role* (the type of the left and right terms), and *else*. This continues for every node throughout the AST. It is worth noting that some nodes do not have any identifier or context associated with them. In that case, nothing is recorded for the word embeddings learning task.

With the rule sets we have created for each node type we have produced fairly high quality word embeddings using the JDK8 library and the access control library Apache Shiro that achieved a state-of-the-practice lowest average loss of 2.69 and 3.17, respectively, which is a perplexity of 14.8 and 23.81, respectively; the perplexity is calculated using base e instead of 2 as Tensorflow's cross-entropy is calculated using natural logarithms. Figure 4 shows embeddings for the 99 most common code elements in Apache Shiro. The embedding vectors were reduced from a 300 dimensional space to 2 dimensions so they could be visualized. We can see a number of good clusterings in the graph. We can see that *checkPermissions* is together with *checkPermission*, *checkRoles*, and *getCredentials*, and in the original vector space the nearest points also include *HashedCredentialsMatcher*, *assertAuthorized*, *AbstractSessionManager*, and *getObjectPermissions*. All of these classes and methods perform similar functions, or make use of, the *checkPermissions* method. We can also see that *isPermitted* is clustered with *isPermittedAll* and *hasRole*, and in the original vector space it is also nearest to *DefaultPasswordService*, *setAuthenticated*, and *getRoles*. For JDK8, we also found good clusterings. For example, if was nearest to *AND* (or &&), *OR* (or ||), *LOGICAL_COMPLEMENT* (or !), and ? (or the *conditional* statement). This clustering makes sense as the behavior of the if statement is heavily reliant on boolean operators and the *conditional* is simply a shorthand version of if. The *PLUS* (or +) operator was nearest to *MINUS* (or −), *int*, *long*, *String*, and *STRING*. This clustering is very good as *MINUS* is also an algebraic operator and *PLUS* performs many of its operations on int, long, and String variable types. While there are a number of good clusterings, some methods and classes did not seem to cluster well in the vector space. For example, some getter and setter functions were found quite far from where a majority of getters and setters were clustered. We believe this is due to the limitations in our current model, which we discuss in Section 5.1.

4 FUTURE WORK

We have shown we can obtain adequate, state-of-the-practice word embeddings for source code. Our next step is to perform learning tasks with them that will be able to link code elements to access control policy statements. This will assist us in building an access control model that can be used to verify the system is consistent with its policy. Figure 5 shows an outline of our future steps in this project. The first step in creating these links will be to collect a dataset to train a deep learning network on. This will probably be the most significant challenge, as described in Section 5.2. We have not found a dataset applicable to our research goals, which means we will have to create our own dataset before any learning can be performed. Since it is very difficult to find access control policies for real software systems that are publicly available, we will explore the idea of using the Health Insurance Portability and Accountability Act of 1996 (HIPAA), which dictates government mandated policies and requires general access control policies for the view, modification, and storage of patient data and documents in medical systems. Since most policies are described using natural language, we will not be able to directly link policy elements to code elements. However, formal software specifications can be directly linked to code elements, which can be constructed from the natural language policies. We will couple these HIPAA policy specifications with open source medical software and manually map code elements to policy roles, objects, permissions, etc. in the specifications. We will use these mappings to train a neural network to identify and construct such mappings. After obtaining a set of mappings constructed manually and a set of mappings constructed by the neural network, we will use each set individually to identify all usages of the linked code elements in software and check if any usage violates policy specifications, which would imply a violation of the policy. We will finally compare the performance of the manual and predicted mappings in detecting policy violations.

We will initially focus on role-based access control (RBAC) as it is one of the more popular access control methods. Our first goal will be to perform a classification learning task to identify which code elements represent the roles and permissions in the RBAC policy. For example, the statement, "User accounts for the system will only be created on the correct authority by the system administrator," could be mapped to the code elements in the code snippet from

Node Types	Identifier	Code Elements Used As Context
Array Creation Expression	array data type	"new []"
Array Type	"[]"	array data type
Assignment Expression	assign operator	target and value expressions
Binary Expression	binary operator	left and right term expressions
Boolean Literal	"true" or "false"	"boolean"
Break Statement	"break"	associated loop or switch statement
Cast Expression	cast data type	expression being cast
Catch Clause	"catch"	exception types being caught
Char Literal	"CHAR"	"char"
Class Or Interface Declaration	name	"class" or "interface" and modifiers, interfaces, extension, and members
Conditional Expression	"?"	condition expression and "else"
Continue Statement	"continue"	associated loop statement
Do Statement	"do"	"while" and condition expression
Double Literal	"DOUBLE"	"double"
Enum Declaration	name	"enum" and all modifiers
Explicit Constructor Invocation	"this" or "super"	associated arguments and expressions
Field Declaration	field data type	initializations, assignments, and modifiers
For Each Statement	"for"	"break" and "continue" if present and variable and iterable types
For Statment	"for"	"break" and "continue" is present and initialization type, condition expression, and update expression
If Statement	"if"	condition expression and "else" if present
InstanceOf Expression	"instanceof"	expression and instance type
Integer Literal	"INT"	"int"
Long Literal	"LONG"	"long"
Method Call Expression	name	arguments
Method Declaration	name	modifiers, return type, parameters, thrown exceptions, and method body
Null Literal	"null"	"void"
Object Creation Expression	object data type	"new", arguments, and anonymous class body if present
Primitive Type	primitive data type	associated expressions or declarations
Return Statement	"return"	associated expression
String Literal	STRING	"String"
Super Expression	"super"	associated expression
Switch Entry Statement	"case" or "default"	"switch" and associated label expression if present
Switch Statement	"switch"	selector expression and switch entry statements
Synchronized Statement	"synchronized"	associated expression
This Expression	"this"	associated expression
Throw Statement	"throw"	associated expression
Try Statement	"try"	resource expressions, catch clauses if present, and "finally" if present
Unary Expression	unary operator	associated expression
Variable Declaration Expression	variable data type	modifiers and initialization expression if present
Void Type	"void"	associated expressions or declarations
While Statement	"while"	"break" and "continue" if present and condition expression

Table 1: Node Type Identifiers and their associated Context

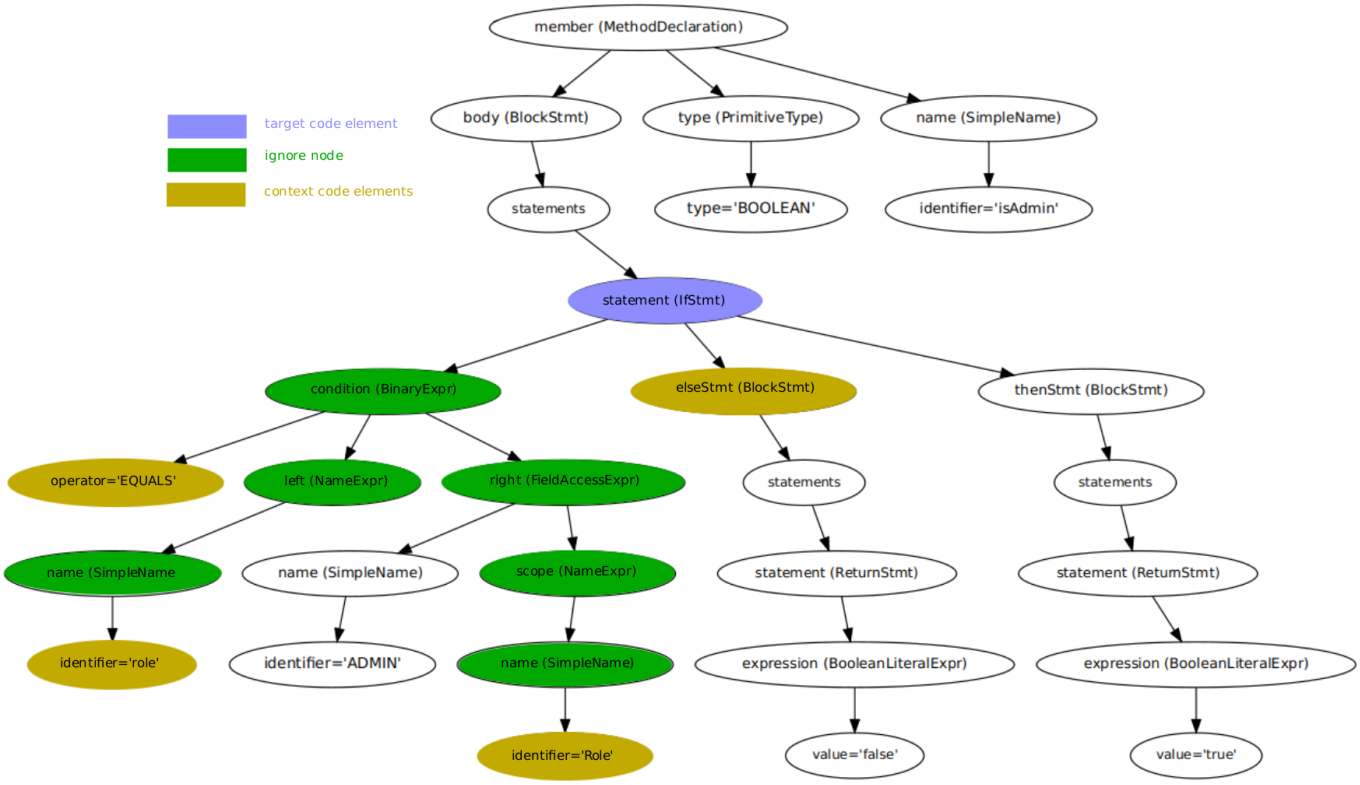

Figure 3: Example AST of a Function *isAdmin* and the Context Extracted for *IfStmt*

Apache Shiro in Figure 6, where *if*, *hasRole*, and *administrator* define the role and what would be the *if* statement body defines the account creation call (or data access and modification). Finally, we will be able to use these mappings to determine if any part of the RBAC policy is violated.

5 RESEARCH ROADMAP

In access control there are many different learning tasks that can be utilized in current and future research, especially if those learning tasks can be applied to source code, such as: generating formal access control specifications described in natural language policies, generating targeted testing of access control policies, policy verification or policy violation detection, access control vulnerability detection, and more. In this section we describe some of the major problems of deep learning on source code.

5.1 Word Embeddings

As stated above, we believe our word embeddings can be improved. The quality of source code word embeddings directly and drastically affects the performance of neural networks on any learning task dealing with source code. The primary detriment to the quality of our embeddings is properly utilizing defined context for method declarations. For all other node types, the context provided has two common properties. First, each individual code element being used as context for a target code element is independent of the other code elements in the context. For example, consider a class declaration's context which includes modifiers, interfaces, extensions, field names, private class names, and method names. Each of these code elements give meaning to the class, but that meaning is not dependent on any other code element in the context. That is, for example, the meaning that a method name gives to the class has nothing to do with the meaning that a modifier gives to the class. This is different for methods in that a method body gives meaning to the method as a collection of statements that are dependent on each other and not as individual code elements. That is, changing even a single code element in the method body (such as changing an if statement to a while statement) could easily change the entire purpose and functionality of that method; in the same way, the relative meaning given to the method's word embedding should also reflect this possible drastic change, which would be difficult by considering this particular context independently. Second, in being independent, code elements for other node types do not derive any meaning from the order that they are listed in the context. For example, a class declaration should produce roughly the same word embedding regardless of the order its methods were defined in. However, for a method body the order of statements can matter a great deal. Research is needed to design more effective embedding methods. Possible methods to consider may be based on recurrent neural networks [4, 14], other strategies to gather more appropriate

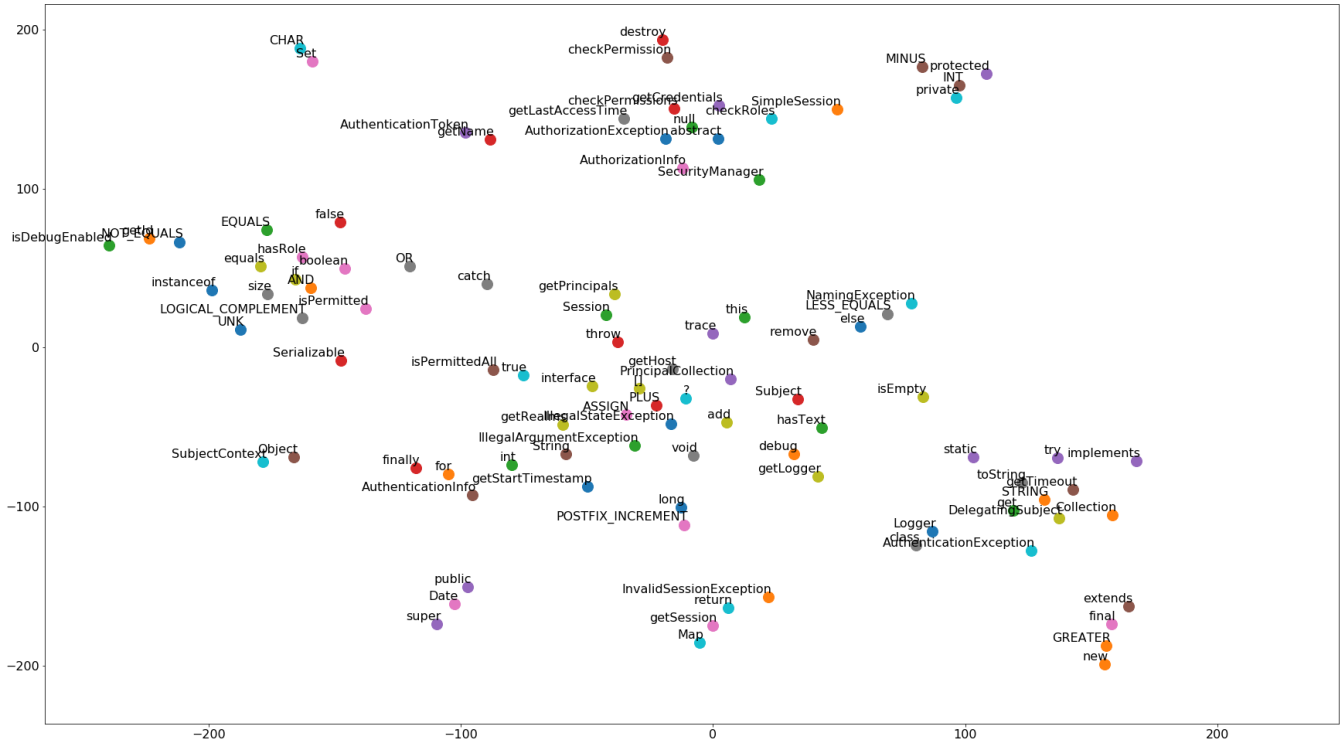

Figure 4: Apache Shiro Core Embeddings For 99 Most Common Code Elements

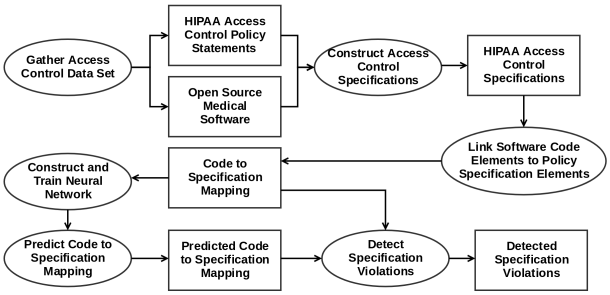

Figure 5: Map of access control policy violation detection future work

```
//get the current Subject
Subject currentUser = SecurityUtils.getSubject();

if (currentUser.hasRole("administrator")) {
    //show a special button
} else {
    //don't show the button
}
```

Figure 6: Code snippet example of checking for administrator role

context for target code elements, or alternate ways for word embeddings to represent collections of code elements at the phrase [4], statement, or path [2] levels.

5.2 Datasets

One of the most significant challenges with any deep learning task is acquiring enough data to train a neural network and test its performance. Datasets have been created for many deep learning tasks, such as the MNIST database of handwritten digits [8] for image classification, many corpora for sentiment analysis from different social media or review websites, and more. Since the idea of performing deep learning tasks directly on source code is still very new with limited literature available, we know of no dataset that exists for any kind of learning task related to deep learning on source code. In current research, most papers automatically generate a dataset to evaluate their neural networks, however, there are problems with creating such datasets.

One of the most popular techniques includes code prediction, where words are stripped from a code snippet and then predicted by the neural network. In most cases, though, the code prediction was performed by removing code elements at the end of statements or method bodies. This tailors a very general problem of predicting the next code element at any point in code to a very specific situation, giving the network an advantage and bias in how well it performs. Further, the next code element chosen by a programmer is based on a number of factors that are not directly linked to code elements,

such as personal coding preferences, learned coding conventions, the current line of logical reasoning, and more.

Vulnerability detection has also been performed, where a static analysis tool is used to annotate whether a code snippet is *buggy* or *clean* [5]. In this case, by using a static analyzer to perform the annotation, the neural network will only learn how to perform like the analyzer and will misclassify code snippets in the same way as the analyzer does. It is also important to note here that not only should the datasets be large, but they should also be robust. In this example of vulnerability detection, even if the dataset was labeled manually, there still may be an inherent problem in only having the two labels of *buggy* or *clean*. Consider the vast number of types of bugs that can appear in source code. Each of these types of bugs can be very different from each other and can appear in code many different ways. It will likely be difficult for a single neural network to encompass all kinds of bugs using a single *buggy* label. That is, the disparate differences between types of bugs can make it difficult for a neural network to find a few patterns, or sets of weights, that includes all different types of bugs and distinguishes these bugs from clean bug-free code. Further, the ultimate goal of vulnerability detection is not only to determine if a bug exists, but also what type of bug is contained in the code. If a dataset only contains the labels *buggy* and *clean*, then a new dataset would need to be created (or the old dataset would need to be updated) with the labels that distinguish those different types of bugs. Therefore, it would be advantageous for the dataset to be robust. In the case of vulnerability detection, a possible solution may be to split the *buggy* label into a small set of bug categories that will allow a neural network to more easily find division in the data it is given.

5.3 Neural Network Construction

The final problem we will discuss is a problem that is significant to all research that utilizes deep learning, which is what kind of neural networks and neural network configurations work best for the specific learning tasks being performed? The particular construction of a neural network for different learning tasks on source code and in access control problems can affect its performance a great deal, and is an area of great variability and creativity.

There are different types of neural networks, and what kind of network is being used for a particular learning task can greatly influence performance. For example, convolutional neural networks (CNN) are adept at identifing the most important features in data using filters, and recurrent neural networks (RNN) are able to process data series and sequences into a single representation. Both networks have been shown to perform very well in text learning tasks, and we believe should be easy to translate to source code tasks.

The number of hidden layers and weights are an important balance in constructing neural networks. In general, the more layers and weights there are in a network, the greater ability a network has to learn. That is, the more information the network is able to store and leverage in its learning task. However, the more layers and weights in the network, the longer it takes for the network to learn, sometimes taking days, weeks, or even months to complete. Further, there are many other configurations a network contains, all of which affect the performance of the neural network in different ways, such as different kinds of activation functions, the form of the input being passed in, the loss function used, the size of the batches of data being learned over at a time, the number of times to learn over the entire dataset, and more. Experiments are needed to study which configurations perform best for source code learning tasks.

6 CONCLUSION

In this paper, we discussed the feasibility of deep learning on source code, its potential contributions to research in access control, and some of the major open problems that will need to be addressed. We described how we plan to use deep learning to perform access control policy verification, as well as our first steps toward creating quality word embeddings for source code elements. We believe deep learning has much to offer the access control community, especially when utilized in learning tasks on source code.

ACKNOWLEDGEMENTS

We would like to thank MSI STEM Research & Development Consortium (Grant Award #D01_W911SR-14-2-0001-0012), the CREST Center For Security And Privacy Enhanced Cloud Computing through the National Science Foundation (Grant Award #1736209), the National Science Foundation through their Early-Concept Grants for Exploratory Research (Grant Award #1748109), the National Science Foundation through their Faculty Early Career Development program (Grant Award #1846467), and the National Security Association (Grant Award #1804556) for their contributions to this project.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 404–419.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[5] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[6] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[7] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[11] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 547–553.

[12] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[13] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.

[14] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[15] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[16] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 334–345.