CODE ELEMENT VECTOR REPRESENTATIONS THROUGH THE APPLICATION OF NATURAL LANGUAGE PROCESSING TECHNIQUES FOR AUTOMATION OF SOFTWARE PRIVACY ANALYSIS

by

JOHN HEAPS, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
August 2020

DEDICATION

I would like to dedicate this dissertation to my family, friends, colleagues, and mentors for all their support. I truly would not have come this far without them.

ACKNOWLEDGEMENTS

I would like to thank the CREST Center For Security And Privacy Enhanced Cloud Computing (C-SPECC) through the National Science Foundation (Grant Award #1736209) for their continued support and contributions to my research.

August 2020

CODE ELEMENT VECTOR REPRESENTATIONS THROUGH THE APPLICATION OF NATURAL LANGUAGE PROCESSING TECHNIQUES FOR AUTOMATION OF SOFTWARE PRIVACY ANALYSIS

John Heaps, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professors: Jianwei Niu, Ph.D. and Xiaoyin Wang, Ph.D.

The increasing popularity of mobile and web apps has prompted an increase in the collection and storage of personally identifiable information of app users, causing users to be at continually greater privacy risk if that information is misused or mishandled. In order to reduce such risk, software must be in compliance with privacy policies, which detail how privacy information is collected, stored, and maintained. The current state-of-the-art in determining privacy compliance is through static analysis techniques such as model checking or pattern-based detection, but these techniques lack both automation and generalizability, suffering from limitations such as: the state explosion problem, conservatism, and the manual definition of models, specifications, or patterns. Deep learning models and approaches have been shown to help solve similar limitations in other problem domains, and may be adaptable to privacy policy and software analysis. Being written in natural language, privacy policies can be immediately applied to deep learning natural language processing. Further, it has been asserted that code can be treated and processed like a natural language as it exhibits behaviours and properties analogous to natural language. However there are many obstacles in the application of deep learning to code, such as: the complex syntactic structures of code, the constant definition of new code elements, and increased severity of the data sparsity problem. This work proposes a novel semantic learning approach that can help overcome such obstacles by taking advantage of the equivalence relationship between code elements and their declarations and definitions. The approach is implemented by creating a plugin to perform textual pre-processing and preparation unique to code and constructing deep learning models for producing code element vector representations for processing code elements in deep learning tasks. The models are shown to produce quality vector representations for code elements and are applied to a deep learning task to predict access of privacy information by software.

TABLE OF CONTENTS

Acknowledgements . . . . **iv**

Abstract . . . . **vi**

List of Tables . . . . **xi**

List of Figures . . . . **xii**

Chapter 1: Introduction **1**

1.1 State of the Art 4

1.1.1 Finding A Language Model For Deep Learning On Code 4

1.1.2 Deep Learning Models Utilizing N-Gram Approaches For Code Analysis . 5

1.1.3 Deep Learning Models Utilizing Abstract Syntax Trees For Code Analysis 6

1.2 Challenges and Limitations 7

1.2.1 Static Analysis 7

1.2.2 Deep Learning on Natural Language 9

1.2.3 Deep Learning on Code 10

1.3 Approach 11

1.4 Contributions 12

Chapter 2: Background **14**

2.1 Deep Learning 14

2.1.1 Artificial Neuron 14

2.1.2 Feed-Forward Neural Network 22

2.1.3 Recurrent Neural Network 24

2.2 Word Embeddings 27

2.2.1 N-Gram 29

2.2.2 Word2Vec 30

2.3 Code as a Natural Language 31

2.3.1 On the Naturalness of Software 32

Chapter 3: Overview 34

Chapter 4: Privacy Compliance with Static Analysis 36

4.1 Verifiable Assume-Guarantee Privacy Specifications for Actor Component Architectures 36

4.1.1 Privacy Policy Specification 37

4.1.2 System Framework and Specifications 38

4.1.3 Model Checking and Evaluation 42

4.1.4 Limitations 43

Chapter 5: Automation of Privacy Policy Analysis 45

5.1 Learning Ontologies from Natural Language Policies 45

Chapter 6: Learning Model for Code Element Vector Representations 53

6.1 Vector Representations Using Current Natural Language Processing 53

6.2 Semantic Vector Representations Using Code Element Definitions 57

6.2.1 Pre-Processing 61

6.2.2 Learning Model 62

Chapter 7: Automation of Software Privacy Analysis 68

7.1 DICTA: Dynamic-Analysis-Induced Information-Type Classification of Call Traces in Android Apps 68

7.1.1 Learning Model for Stack Trace Prediction 72

Chapter 8: Evaluation 77

8.1 Vector Space Analysis 77

8.2 Privacy Compliance Results 79

8.3 Discussion 80

Chapter 9: Conclusion 81

9.1 Future Work 81

9.1.1 Defect Detection and Measuring Software Reliability 81

9.1.2 The Novel Learning Approach for Natural Language 88

Bibliography 89

Vita

LIST OF TABLES

5.1 Experiment 1 Results 50

5.2 Experiment 2 Results 51

6.1 Node Type Identifiers and their associated Context 58

7.1 User Profile 72

7.2 Results of applying the different models to the testing set 76

LIST OF FIGURES

2.1 M-P Neuron Functions 15

2.2 M-P Neuron 15

2.3 Removal of θ to a bias term 18

2.4 Perceptron 18

2.5 Perceptron Learning Algorithm 19

2.6 The Step Function 20

2.7 The Sigmoid Function 21

2.8 The Sigmoid Neuron 21

2.9 Feed-Forward Neural Network 22

2.10 Recurrent Neural Network 25

2.11 Unrolled Recurrent Neural Network 26

2.12 Recurrent Neural Network Cell 26

2.13 Word Embedding Vector Space 28

2.14 Identifying N-Gram Training Context 29

2.15 Identifying Word2Vec Training Context 30

2.16 English cross-entropy vs. Java Code cross-entropy 32

4.1 Framework 39

4.2 EMRS Decomposition 41

5.1 Ontology Learning Architecture 48

5.2 Phrase Modeling Layer 49

6.1 Word2Vec Code Element Embeddings 54

6.2 Vector Representations for the 35 Most Common JDK8 Code Elements . . 54

6.3 Vector Representations for the 99 Most Common Apache Shiro Code Elements 59

6.4 Learning Model 63

6.5 Variable Embedding Model 64

6.6 Method Embedding Model 65

6.7 Class Embedding Model 66

6.8 Vector Representations for the 45 Most JDK8 Code Elements 67

7.1 Example Stack Trace 70

7.2 Stack Trace Learning Model 74

8.1 Clustered Vector Representations for the 99 Most Common Apache Shiro Code Elements 78

8.2 Clustered Vector Representations for the 45 Most JDK8 Code Elements . . 78

9.1 Bug Pattern Framework 83

CHAPTER 1: INTRODUCTION

A recent surge in concern over the management of user privacy data by mobile and web apps has been seen in recent years. As app owners continue to increase their collection and storage of more users' personally identifiable information, we have seen more cases of information mismanagement, exposing app user to extreme privacy risk. There have been a number of recent changes and updates to privacy laws [2, 18, 59] to help protect users and mitigate such privacy risk. Further, app owners are now required to keep a privacy policy to detail what privacy information their app collects, stores, and maintains, allowing users to understand the privacy risks each app contains. By using static analysis techniques, privacy compliance between software code and such laws and policies can be verified, identifying privacy violations and minimizing privacy risk. However, it can difficult to automate and generalize static analysis techniques, making the ability to ensure software compliance over most apps impractical.

Some of the most common static analysis techniques use pattern detection and model checking. These techniques search through code and use code patterns (i.e., a recognizable sequence of code elements that correspond to a previously defined standard or example) and model specifications (i.e., a formal logical formula that describes a desired or unwanted behavior of a system) to perform its analysis. If a code snippet or state path within the system matches one of those patterns or violates one of the specifications, the occurrence is recorded and reported. Developers are then able to check the indicated code or states that are found there. This easily allows for the definition of new patterns and specifications in static analysis, as a pattern or specification simply needs to be designed and implemented for it.

While such tools are the state-of-the-art and are widely used, there are a number of limitations that have been identified for static analysis techniques: static analysis tools require patterns, models, and specifications to be manually designed and implemented by developers to perform analysis tasks; static analysis is often conservative, since there often exists no dataset to perform proper testing it is then difficult or impossible to determine false negatives in its results, meaning it is usually better to have a high false positive rate (the results of which can be easily checked) than have any false negatives (as they are unable to be checked without examining the entire software system outside of the given results); scalability is an issue specific to model checking as it must check every possible state along all possible call paths (or traces), which leads to the state explosion problem. Overall, there is a need for greater generalizability and automation of manual tasks in static analysis.

A solution to many of these limitations may lie in the application of deep learning. Deep learning does not need patterns or specifications to be provided to it, as it learns features and patterns relevant to its learning task on its own. Since it learns directly from code examples, it should be naturally less conservative. It also does not exhibit the state explosion problem as it simply analyzes words directly, similar to pattern detection. Those deep learning techniques most relevant to privacy compliance verification are those related to natural language processing (NLP).

Deep learning on natural language works by creating a vector representation for every word in a vocabulary to represent a language model. The current language model used to represent natural language text is a statistical model based on the probability of occurrence. This same approach has been applied to code as well. However, such a language model does not represent code elements as well as normal natural language. Complex syntactic structures make the probability of occurrence of a code element more random than that of natural language. It is also more likely that new code elements are encountered after training due to the constant definitions of new method and variables, which cannot be handled since a large amount of occurrences are needed during training to learn proper probability representations. Further, it is more intensely subject to the data sparsity problem since many code elements defined in a system do not occur often enough to learn quality representations and there is often no way to acquire more occurrences such as in natural language (where simply incorporating more documents into training usually leads to more word occurrences).

In this work, a new semantic learning approach is proposed to better represent code elements to improve the automation and generalizability of privacy compliance with deep learning. This will be done by utilizing the equivalence relation between a code element and its definition. Further, a novel deep learning architecture is described which will allow for the processing of code elements that have never been encountered before and will greatly mitigate the data sparsity problem. Finally, novel variations on how to embed code for deep learning and perform privacy compliance learning task predictions are described.

Problem

There are many static analysis tools and techniques that are used in the development and quality enforcement of software. These tools have been successful and helpful in identifying problems early in production, helping ensure security and privacy risks are minimized and saving stake holders time and money. However, static analysis has some major limitations, such as: the need to manually define patterns or specifications to perform detection and enforcement tasks, the inability to handle new or undefined patterns or specifications, the state explosion problem, and that its conservatism often leads to results with high false positive rates. By applying deep learning techniques and methodologies to code, some of these limitations may be mitigated. However, a new learning model and more sophisticated encoding and modeling techniques will likely be needed to apply deep learning effectively. These contributions will allow static analysis to be more generalizable and traditionally manual tasks to be automated.

Thesis

Deep learning techniques can be effectively applied to static code analysis by introducing a new semantic learning approach that will be able to overcome the limitations of traditional static analysis and current deep learning techniques by utilizing the equivalence relation between code elements and their declarations and definitions.

1.1 State of the Art

1.1.1 Finding A Language Model For Deep Learning On Code

The following works helped lay the foundation of research into the subject of deep learning on code by showing that a language model can be used to represent code elements as real-valued vectors that are usable and learnable by deep learning models.

Gabel et al. [23] is one of the earliest contributions to considering using deep learning for code analysis. The paper explores the level of uniqueness of source code across software files and projects. That is, how lexically or syntactically similar or redundant source code is. The study performs many versions of clone detection at different granularities (i.e., different code element lengths). It shows that at lower level granularity, code is not unique and code redundancy can reach 100% redundancy even at granularity of code element length 6; but at granularity of length 20 the uniqueness has risen greatly and redundancy is about 60% or below. This was one of the first steps in theorizing that because code is generally not unique, much of it exhibits similar patterns and behaviors both within-project and cross-project. Therefore, those patterns may be learnable (instead of being manually defined) for code analysis purposes.

Hindle et al. [32] is considered, by most, to be the seminal paper for research into utilizing deep learning on code and was inspired primarily by Gabel et al. [23]. The paper explores the idea of code having a “naturalness” similar to natural language. It builds a basic statistical language model for both English and code corpora. Then cross-entropy and perplexity measures are used to define the “naturalness” (or “unsuprisedness”) of the model. The models show that code appears to closely exhibit the same behaviors and trends as natural language, and even suggests code may be more “natural” than natural language. It concludes that because code is natural, the same deep learning techniques used to analyze and process natural language should be adaptable to use in analyzing and processing code. That is, a learning model should be able to represent code just as well as (or possible better than) natural language.

1.1.2 Deep Learning Models Utilizing N-Gram Approaches For Code Analysis

These works learned vector representations using the same (or very similar) techniques and models as normal natural language. The immediately surrounding code elements of a target code element were used as context for word prediction to determine its vector representations.

Hellendoorn et al. [30] 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. Their work implied 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 which only uses the surrounding code elements to determine vector representations and perform the code prediction. This ignores the structure of code which could have been utilized by the learning algorithm to give better results.

White et al. [79] learns a software language model using an RNN. They discuss motivations for using deep learning and explain the construction of their RNN model. The RNN is compared to a "state-of-the-practice" n-gram algorithm and shows the RNN outperforms it in 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.

Huo et al. [37] utilized bug reports to perform bug detection at the source file level. That is, given a bug report and a source file, does the file contain the bug described by the bug report? This is done by first encoding each word and code element using one-hot encoding. Then, utilizing a convolutional neural network (CNN), natural language representations are learned for a bug report and source code file. The representations are then integrated using a fully connected layer. A comparison between the natural language and programming language CNN (NP-CNN) model and six other models showed NP-CNN outperformed all of them in mean average precision at 55.7%.

Dam et. al. [19] performs vulnerability detection at the file-level. That is, if a source file contains at least one vulnerability, it is reported as "vulnerable" (and otherwise "clean"). This is determined using a combination of vectors at the "local" and "global" level. The local level is determined by using a type of RNN, long-short term memory (LSTM), to encode each individual method contained in the source file into a single vector representation followed by a statistical pooling of all the encoded methods. The global level uses all instances of code elements throughout the corpus to create vector representations. For each source file, the vector space is then grouped into k regions (using k-means clustering) and then a code element frequency count is performed by counting which region each individual code element in the source file falls into. This creates a global vector (each element/dimension representing a region with the frequency counts for each region). These two local and global vector representations are input to a classifier to determine vulnerable or clean. The model achieves 89% accuracy in determining if a give source file is vulnerable or clean. This model is only performed at the file-level, which is not desirable in many cases. Seeing how this kind of model could be adapted to the method (or statement) level and how well it performs would be more applicable to most problems.

1.1.3 Deep Learning Models Utilizing Abstract Syntax Trees For Code Analysis

These works learn vector representations using the Abstract Syntax Tree (AST) of the source code instead of the source code directly. This allows for some advantages, such as: being able to ignore syntactic symbols (e.g., open and close braces, semicolons, etc.), being able to identify different code blocks and structures more easily, etc.

Peng et al. [60] describes building vector representations 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 vector representations, 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 4 programs part of this experiment. While it is interesting that the model was able to learn code and relate it to an ID, it is unclear how this might perform in other larger and more complex learning tasks, as mapping a piece of code to a randomly assigned ID does not represent normal code analysis tasks.

White et al. [78] performs clone detection using deep learning. They model the lexical code features and structural code features separately using Recurrent NN and Recursive NN, respectively. The lexical features of the code are processed in order (i.e., predict the next token) to define a learned vocabulary. 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 clone detection on 8 GitHub Java projects, 6 of which had a precision above 90% and the remaining 2 had precision scores just below 60%. While some results show promise, the lack of other metrics being reported make it difficult to fully understand the effectiveness of the model.

Alon et al. [5] performs function naming using paths (i.e., code traces) of the function declaration and body. They encode each path as a vector representation which is then learned over all given method. While this showed good results for most function naming, since paths through the code are being used for learning, the approach is subject to the state explosion problem.

Tai et al. [70] describes a tree LSTM neural network. The LSTM nodes are able to take in multiple child node inputs and has multiple forget gates.

1.2 Challenges and Limitations

1.2.1 Static Analysis

There are a number of limitations of static analysis techniques that can negatively impact the performance of static analysis tools and approaches.

Static analysis is rarely generalizable and is often highly prone to software decay. It uses patterns or specifications to represent and detect aspects of software code. It requires that these patterns or specifications be manually defined and updated for static analysis to perform those tasks. However, it is usually difficult to maintain such quality and robustness for patterns and specifications for static analysis to be effective long-term. For example, for software bugs, each individual type of bug must have a separate pattern or specification defined for it. To put in perspective the number of these definitions needed, currently the Common Weakness Enumeration (CWE) [54] (one of the largest repositories of known software bugs) has recorded 808 different types of bugs found and reported in software code. Further, each of the 808 different types of bugs can emerge in software in different variations, with each variation requiring a separate pattern or specification to be manually defined for it. Most bugs can also appear in multiple different programming languages, however most patterns or specifications are not able to be applied across different programming languages without being redefined or refactored. This ultimately leads to thousands of different patterns or specifications needed to cover all possible bug occurrences. Further, new types of bugs and different versions of known bugs are discovered and reported often, usually requiring patterns, specifications, and techniques to be manually updated and refined. This is all further exasperated by the continued modification and addition of new features of programming languages. In most static analysis cases it is infeasible for researchers or developers to define and maintain all current and future patterns or specifications.

It can be difficult to find proper datasets for static analysis problems. Static analysis is often in need of large datasets for benchmark and tool analysis purposes. In the current literature, some datasets exist, but most are small or only have very generalized labels (e.g., for bug detection, only labeled as "buggy" or "clean" instead of labeling the specific type of bug). The lack of large, well labeled datasets makes it difficult to determine the quality of static analysis tools and algorithms. The current practice is to either use one of the existing datasets (which can be too small or not well labeled to determine an accurate measure of quality) or to simply run the tool or algorithm on some selected projects or systems and then manually check the results (in which case false negatives and true negatives are either very difficult or not possible to determine).

Static analysis is usually very conservative. A by-product of a lack of good datasets is that it can be difficult to determine the true and false negatives of the results of static analysis tools. Therefore, it is a general practice to define patterns or specifications as generally (and with as many variations) as possible. This conservatism usually leads to detecting many false positives since it is usually better to have a high false positive rate (and be able to easily determine which code segments are false positives) than any false negatives (since they will be missed as they cannot be detected without manually, rigorously exploring all code in the project or system).

Finally, static analysis is not always scalable. Some static analysis techniques and tools require an exploration of all possible paths or states in a program or project. Such an exploration is subject to the state explosion problem which makes such exploration infeasible for a program or project of even average size.

1.2.2 Deep Learning on Natural Language

Deep Learning on software code is approached as a natural language problem. This is somewhat intuitive, as software code is primarily composed of text. The code elements represented by this text cannot be directly processed by deep learning algorithms (as calculations must be done on numeric data). Therefore, these code elements are often represented as real-valued vectors using similar techniques as deep learning on natural language. While this allows code to be processed by deep learning algorithms, it also implies that the traditional problems and limitations of deep learning on natural language are inherited.

Deep learning on natural language must have a large dataset to learn over. In practice, this is simply a very large corpus (usually many thousands of documents as least) of preprocessed text. This large amount of data is needed to learn vector representations that represent the language model. When performing deep learning on code, a similarly large code base will be needed to properly learn a language model.

Deep learning on natural language is often plagued by the data sparsity problem. That is, if a word is encountered that is not commonly used in the domain of the provided corpus, then it is difficult to learn a quality vector representation for the word since there will not be enough instances of the word for the model to train over. Similarly, if there are rarely used code elements in a provided code base, then proper vector representations for the code elements will be difficult to learn.

Finally, after learning a vocabulary of vector representations, if a new word is encountered during subsequent learning or processing tasks, the word cannot be processed since no embedding exists for it. It must either be skipped, be replaced with a generic *unknown* vector representation, or have a vector constructed for it (usually derived from the word's "sub words". In the same way, if during software privacy analysis a code element is encountered that is not in the list of learned vector representations, it cannot be processed by the deep learning model.

1.2.3 Deep Learning on Code

While deep learning on software code has inherited the problems and limitations of natural language, code also has its own unique properties that provide additional challenges.

The syntactic structures of code are far more complex than normal natural language. For example, natural language does not have recurrence (or "loop") or decision (or "if") statements, method and mathematical expressions are rarely used in everyday speech, new words are not being constantly defined and immediately implemented, etc. These new syntactic elements need to be addressed during analysis as they may need more complex processing than the training algorithms used for normal natural language.

When analyzing a new system, there are usually many new method and variables defined in it. That is, a new program or project will be made up primarily of new (or redefined) methods and variables. The only way for these new method and variables to be included in the already defined vocabulary of vector representations, is to analyze and re-learn the vocabulary with the new code elements before performing the actual learning task. However, this will change the current vector representations in the vocabulary, which in turn will be used in a learning model with weights and nodes that were learned against the old version of the vocabulary. This will likely lead to much poorer results than performance during training. The alternative approach would be to either disclude all new method and variables, or replace them with a generic *unknown* vector representation. However, removing or using a generic replacement vector representation for so many code elements will very likely have a significant negative impact on a deep learning model's ability to produce correct predictions. So it is necessary that these new methods and variables be handled in some other way.

Finally, the data sparsity problem is far greater in code than in normal natural language. Though not true for all method and variables, but in many cases in code, when a new method or variable is defined it may be used a very few times in other parts of the project or system. However, as described above, to find proper vector representations for such methods and variables, many usage occurrences must be collected. Since this is often the case for many methods and variables, it will be very difficult to find proper vector representations for those code elements. Further, for many code elements, we are unable to identify more occurrences outside of a given project or system. That is, if a new code element is declared or defined in a project or system, then that code element only exists in that project or system (unless the system is used as a library in other projects or systems). If there are not enough occurrences to learn a quality vector representation, then there is no way to obtain more occurrences. This is different from normal natural language where all words in a language exists (i.e., can be used) in all documents.

1.3 Approach

To overcome many of the limitations of static analysis and address many of the challenges of deep learning and natural language processing, a novel approach toward constructing vector representations of code elements is described and implemented. This approach takes advantage of the equivalence relationship between a code element name and its declaration or definition. The code element's vector representation is constructed from its declaration or definition, which significantly reduces the data sparsity problem (as only a single declaration or definition is necessary to produce a vector representation) and allows for the processing of new code elements not encountered before (as a vector representation can be constructed for it on-the-fly if a declaration or definition exists for it).

In order to process and learn code element vector representations effectively, some code pre-processing is necessary. The most important issue handled by the pre-processing is the renaming of certain code elements. These code elements include: method names, method calls, field variables, local variables, class names, and reference types. During natural language processing, a vocabulary is built of known words that are linked to their respective vector representations. One of the requirements of the vocabulary is that all words (or code elements) entered into it be unique. This is because when assigning vector representations to words during a learning task, if the vocabulary contains multiples of the same word (each linked to a different vector representation) then there is no way of determining which vector representation to use during the learning task. However, due to scoping, inheritance, and reassignment rules of most programming languages, it is possible that many different code elements can share the same name. Therefore it is important to pre-process code element names to ensure that they are unique when they are entered into the vocabulary.

Once pre-processed, the code element vector representations will be constructed primarily using recurrent neural network (RNN) models. The models predict a vector representation directly instead of performing a classification prediction learning task that is normal to state-of-the-art natural language processing techniques. By predicting the vector representation directly, this allows new vector representations for new code elements to be created and used on-the-fly instead of requiring all vector representations to be pre-trained in advance. This will allow for greater versatility in analyzing new systems and programs.

1.4 Contributions

Novel Learning Approach for Code Element Vector Representations

A novel learning approach is presented, based on the equivalence relationship between code element names and their declarations or definitions. Implemented for the Java programming language, learning models produces vector representations for classes, methods, and variables. The learning models help mitigate some of the existing limitations in deep learning on natural language: reduced data sparsity problem, the ability to process new words (or code elements), etc. Further, the model predicts vector representations directly, so is able to produce vector representations on-the-fly, reducing the need for a prepared pre-trained vocabulary.

IntelliJ Plugin for Code pre-processation

A plugin for the IntelliJ IDE was created to pre-process and extract code elements with their attached declarations and definitions. This is then to be used to learn a model to construct vector representations for code elements. The pre-processing includes fully qualifying the names of code elements to ensure easy identification in the vocabulary during training, code element tokenizing, and on-the-fly construction of vector representations.

Improved Vector Representations of Code Elements

The novel learning approach produces higher quality vector representations for code elements than other current learning model approaches. This is shown by exploring the clustering of vector representations in the code element vector space.

Information-Type Stack Trace Classification with Deep Learning

The novel learning approach is implemented to predict if a given stack trace accesses sensitive privacy data and what information type of data that is. The implementation shows significant improvement over a bag-of-words model implemented for the same problem.

CHAPTER 2: BACKGROUND

2.1 Deep Learning

A group of Machine Learning models and architectures based on Artificial Neural Networks (ANNs), Deep Learning attempts to learn how to model data and perform tasks inspired by how the human brain learns. It has shown to perform some tasks at, or exceeding, that of average human capabilities.

Deep Learning is performed through the construction and learning of a Neural Network. A Neural Network is a series of operations (usually visualized as a directed graph) between Artificial Neurons. It's goal is to predict the correct output of a given input. The Neural Network takes an input, transforms that input through its series of Artificial Neurons, predicts an output based on the transformation, and then updates the values of its Artificial Neurons based on how close its prediction was to the expected output (i.e., the ground truth).

2.1.1 Artificial Neuron

The Artificial Neuron (also known as nodes or units) are the basic building blocks of any neural network or model. The following subsections will describe the evolution of the Artificial Neuron into what is currently used in neural networks today.

McCulloch-Pitts Neuron

Considered by most the first Artificial Neuron, the M-P Neuron (also known as the MCP Neuron) was created by Warren McCulloch and Walter Pitts in 1943 [48]. The approach was based on the assumption that the human brain's decision making process could be modeled by using boolean functions.

Each neuron receives a set of inputs x which are either "excitatory" or "inhibitory" and can have the values 0 or 1 ($false$ or $true$, respectively), contains a summing function g, a decision (or piecewise) function f (also called the activation function) with a threshold value θ, and produces

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad if \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad if \quad g(\mathbf{x}) < \theta \end{aligned}$$

Figure 2.1: M-P Neuron Functions

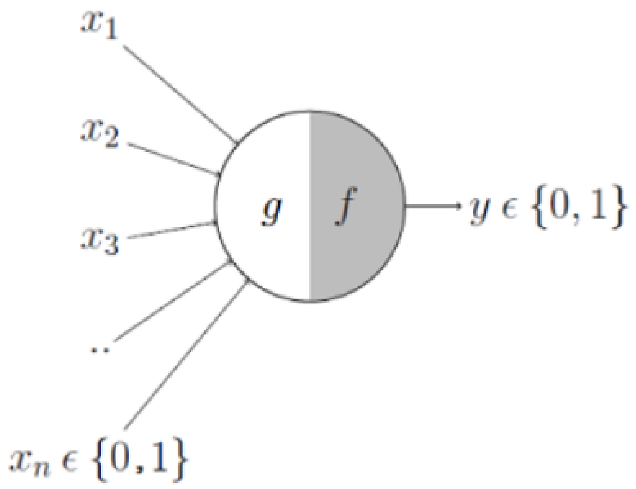


Figure 2.2: M-P Neuron

an output y which can have the value 0 or 1 ($false$ or $true$, respectively). The excitatory input values are summed by g, and f sets the value of y to 1 if the result of g is equal to or greater than θ and 0 otherwise, as shown in Figure 2.1.

As shown in Figure 2.2, each input represents some boolean statement. If that boolean statement is $true$ then it has the value 1, and if it is $false$ it has the value 0. If an input is “inhibitory”, that input MUST be a specific value for the entire neuron to possibly evaluate to $true$ (i.e., if the value is not the specific value, then the neuron is automatically $false$, regardless of any other input values). If an input is “excitatory”, that input will help the neuron evaluate to $true$ through the summing function g.

For example, let us say we want a neuron to determine if we should bring an umbrella with us when we leave for work. Some possible inputs could be *the umbrella is not broken*, *the chance of* *rain is above 50%*, and *it's very hot and sunny outside*. In this case *the chance of rain is above 50%* and *it's very hot and sunny outside* could be excitatory. That is, one may want an umbrella to protect oneself from rain or from intense sun. If one of them is $false$, it does not necessarily mean the entire neuron should evaluate to $false$; just because it is unlikely to rain doesn't mean one won't bring an umbrella if protection from intense sun is still wanted or needed. A θ of 1 would likely be chosen as the threshold. Meaning, one would bring an umbrella (the neuron should evaluate to $true$) if it is the case either (or both) *the chance of rain is above 50%* or *it's very hot and sunny outside*. The input *the umbrella is not broken* could be inhibitory. That is, if the umbrella is broken then the entire neuron should simply evaluate to $false$ regardless of the other inputs since you can't use a broken umbrella and so it would be pointless to bring.

The real power of these neurons is to utilize them in sequences (or networks). For example, the neuron that decides whether or not to bring an umbrella may be part of a larger decision of whether or not to bring a bag. Such a network may decide that if one must bring a certain amount of items θ (umbrella, laptop, change of clothes, files, etc.) then a bag should be utilized to carry everything. In order to determine if a bag is needed, first one must determine how many items one is bringing. Whether or not each individual possible item will be brought to work is likely its own neuron with its own set of inputs to determine if the item is needed (similar to the umbrella). As can be seen, even for such a simple decision, a large network with many input may be needed to account for all possible scenarios and factors that go into making the decision.

A model based on this type of Artificial Neuron has some limitations:

- In this model all inputs are considered equivalent in importance, but in many models different inputs have different levels of importance and are usually modeled using weights (which this model cannot handle). As in the above example, items that are larger, bulkier, heavier, or easily damaged may warrant a bag regardless of the actual number of items one is bringing. If one is only bringing an umbrella and some files a bag may not be necessary, but if one is bringing a change of clothes and a laptop then a bag is very likely desirable.

- In many cases, we want to model input that cannot be represented by a simple boolean statement. For example, if we want to perform decisions based off image processing, simple boolean statements may not be enough as we need to consider colors and shapes within the image.

- In this model, we must select and code in every single input. It is very likely (especially in large networks) that important scenarios and factors may not be considered when constructing the network and so many important inputs will not be included, which will bias the results.

- On a similar note, every threshold must be manually selected. In many cases it is not easy to determine what the optimal threshold is.

- Each individual M-P Neuron is also only able to handle linearly separable decisions. That is, the decision space must be able to be separated (or categorized) into two parts. This is evident in that the function f only separates the space into two parts, 1 for anything at or above θ and 0 for everything else. For example, the XOR operation (given two choices, exactly one of them must be true to evaluate to $true$) is not linearly separable. That is, the neuron should evaluate to θ if either the function g evaluates to below or above θ and 1 only if it evaluates to exactly θ. One would need at least three definitions for f to implement this.

Perceptron

The Perceptron was original proposed by Frank Rosenblatt in 1958 [64] and was then further analyzed and refined by Marvin Minsky and Seymour Papert in 1969 [**?**]. Using the M-P Neuron as a foundation, there are 3 main differences between it and the Perceptron.

The first difference is the addition of weights that are multiplied with the inputs before summing and comparing to the neuron's threshold. This not only allows the input to be weighted, but also allows the inputs to be real numbers (instead of just 0 or 1, $false$ or $true$). The second difference is a mechanism to learn what the "optimal" values for those weights are. The final difference is the idea of removing the threshold θ from the function f and instead re-introducing it as a bias term

$$\begin{aligned} \mathsf{f} &= 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta \\ &= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \mathsf{f} &= 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0 \\ &= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \mathsf{f} &= 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0 \\ &= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0 \\ & where, \quad x_0 = 1 \quad and \quad w_0 = -\theta \end{aligned}$$

Figure 2.3: Removal of θ to a bias term

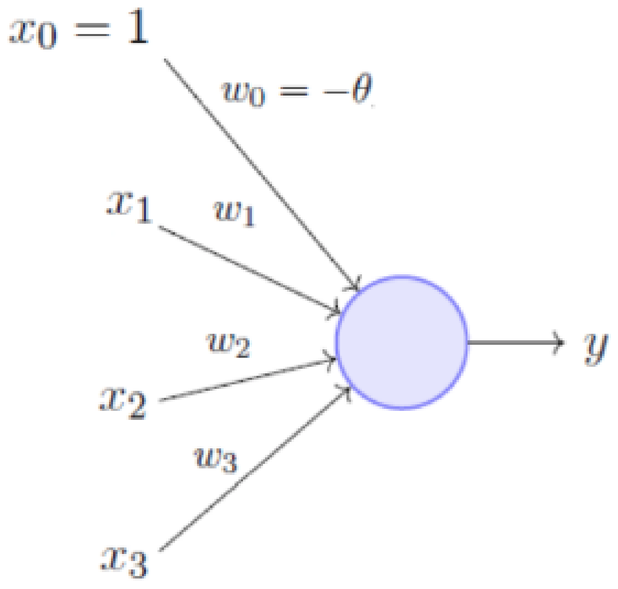
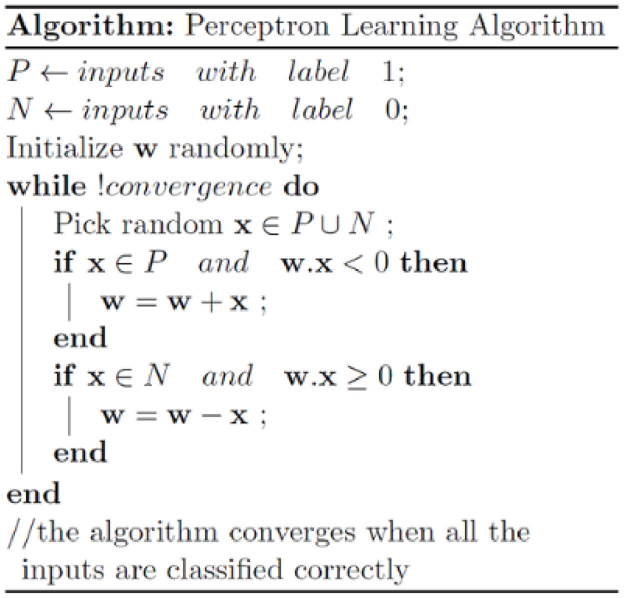


Figure 2.4: Perceptron

in the input. This allows the threshold to also be learned like the weights and removes the need to manually encode them.

Figure 2.3 and Figure 2.4 are the equations and neuron, respectfully, that shows the addition of weights, as well as how θ is removed from the function f and then added as bias to the set of input.

The way the weights and bias are learned is by using a simple algorithm, as shown in Figure 2.5, and data examples. The weights and inputs have been converted into corresponding vectors (e.g., the perceptron from Figure 2.4 would be converted to $vector_x = [x_0, x_1, x_2, x_3]$ and $vector_w = [w_0, w_1, w_2, w_3]$), and the notation $w.x$ is the dot product of $vector_w$ and $vector_x$. The algorithm starts by taking a set of examples labeled with what the neuron should output given an assignment of all the inputs (input x_0 is the bias and should always be 1). A P, or positive, example should output 1 and an N, or negative, example should output 0. Then, initialize all weights to random values. While the perceptron hasn't converged (i.e., it's unable to determine the correct output for



```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow inputs \quad with \quad label \quad 1$;
$N \leftarrow inputs \quad with \quad label \quad 0$;
Initialize $\mathbf{w}$ randomly;
while $!convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```

Figure 2.5: Perceptron Learning Algorithm

all given input examples), choose a random example (or simply cycle through all the examples). If the example should output 1 (i.e., is P), but it would output 0 (i.e., the dot product produces a value less than 0) then change the weights by adding x to w (i.e., vector addition). If the example should output 0 (i.e., is N), but would output 1 (i.e., the dot product produces a value greater than or equal to 0) then change the weights by subtracting x from w (i.e., vector subtraction). This essentially moves the values of the weights back and forth until they define a separation of the inputs in the input space (i.e., all possible valid input combinations) that will allow the perceptron to output the proper result every time. The algorithm has been proved to always converge.

It is worth noting that the learning algorithm is separate from the Perceptron (or any Artificial Neuron) itself. That is, a different learning algorithm can be defined without changing the model. As neural networks have evolved, superior learning algorithms have been introduced.

The Perceptron can still only handle linearly separable problems, just like the M-P Neuron. However, it has been shown that a Multi-Layer Perceptron (MLP) can solve non-linearly separable problems (e.g., the XOR function can be modeled using MLP).

This is a vast improvement over the M-P Neuron and solves the problems of weighted inputs, using real numbers for input, and automatically determining the optimal threshold (or bias); how-

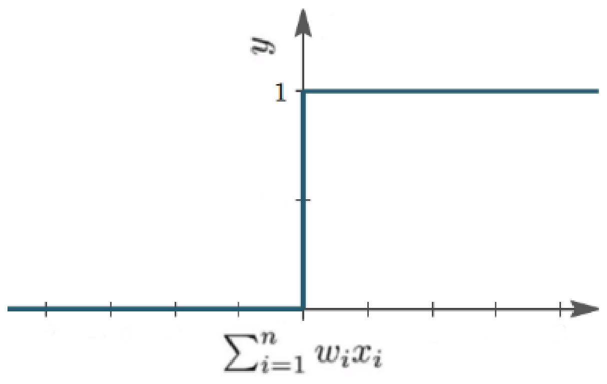


Figure 2.6: The Step Function

ever, there is still a major issue that remains:

- The algorithm only converges based on the example inputs given to it. Given a new labeled example, the Perceptron is not guaranteed to give the correct output.

Sigmoid Neuron

What is now viewed as the "normal" Artificial Neuron, the Sigmoid Neuron is essentially the same model setup as the Perceptron, but the f function from the Perceptron and M-P Neuron has been modified. In the previous cases, f essentially defines a step function. That is, we have linearly separated the input space into two parts, defined by the weight vector of the inputs. When the calculation of the function g is below that line the output is 0, and when it is at or above the line the output is 1. This behavior is defined by Figure 2.6.

However, it is not usually desirable to have such a "hard" cutoff between two options. Further, in most real life scenarios, a decision being made is rarely 100% yes (or 1, or $true$) or 100% no (or 0, or $false$). For example, the choice to buy a car, the choice to see a movie, the choice of what restaurant to go to, the choice of whether or not to bring an umbrella when you go out, etc. are dependent on many factors that may sway one's decision, but rarely do all factors and inputs into the decision align to give 100% confidence in the decision made. That is, while most decisions one makes are made with a sense of certainty, rarely are they made without at least a slight feeling of apprehension or consideration of other possibilities. Therefore, in most cases, it is desirable for a

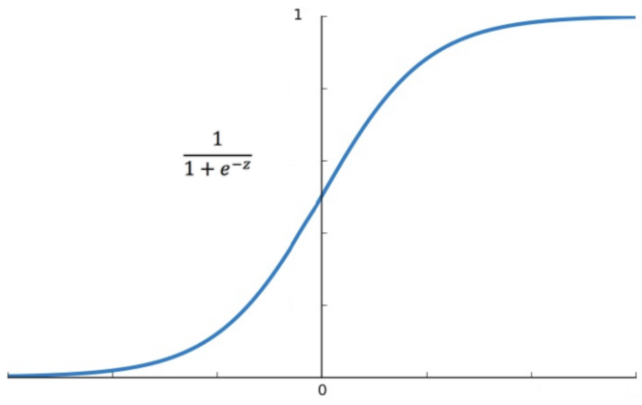


Figure 2.7: The Sigmoid Function

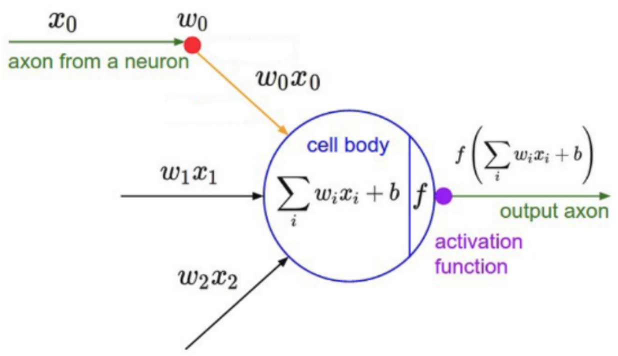


Figure 2.8: The Sigmoid Neuron

neuron to have both a smoother transition between two options and more reasonable probabilities of confidence in the decision being made.

As mentioned earlier, the function f is also called the activation function. In order to change the behavior of the neuron, all that is needed is to change f to a different activation function. In this case, that function is the sigmoid function. It's behavior is shown in Figure 2.7.

As can be seen. This provides a smoother transition between 0 and 1, and provides a probability output instead of an absolute value of 0 or 1. It should be noted that the sigmoid function is not the only possible function that can be used as an activation function (though it is one of the most popular). Other activation functions include: hyperbolic tangent (TanH), rectified linear unit (ReLU), etc. Figure 2.8 shows a generalized Sigmoid Neuron, where f can be any activation function.

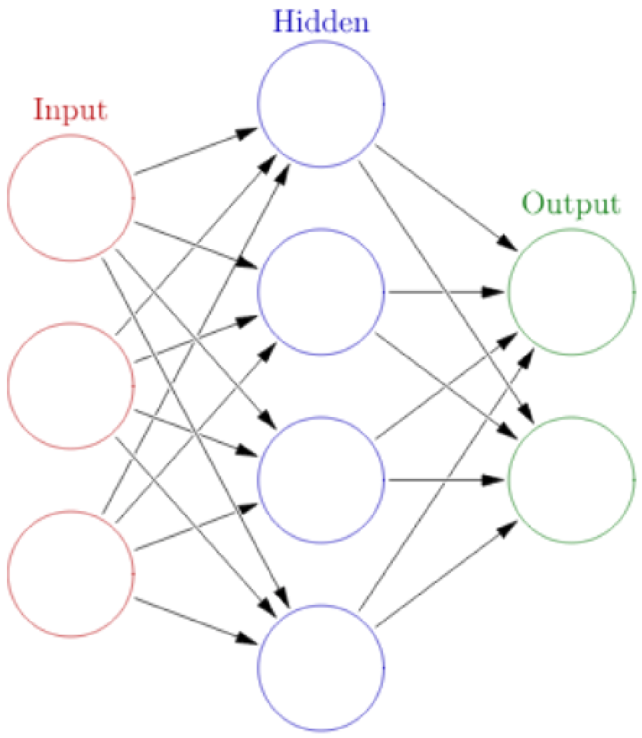


Figure 2.9: Feed-Forward Neural Network

Finally, with the introduction of the sigmoid function, learning the optimal weights and bias become much more difficult than in the original step function of the Perceptron. In many cases complete convergence (i.e., obtaining the correct answer for all given example inputs) is not possible. Therefore, other loss functions are used based on the type of learning task being performed, such as: Mean Squared Error, Gradient Descent, etc.

2.1.2 Feed-Forward Neural Network

The Feed-Forward Neural Network is one of (if not the most) basic type of neural network and set the foundation for what deep learning is today.

The network is based on three types of densely connected layers, with each layer containing a set of Artificial Neurons. These three layers are “Input”, “Hidden”, and “Output”. Figure 2.9 shows a basic representation of a Feed-Forward Neural Network.

Input Layer

The Input Layer is simply the input to the network. The input can represent any kind of real-valued data, such as: image/pixel values, population/demographic data, sales data, etc. The input is usually visualized by showing an input neuron (or input node) for each individual input value to the network. During programming and processing of the network, the input values are usually given in vector or matrix form. In Figure 2.9, the Input Layer shows an input vector with 3 values (which would normally be represented as $[x_0, x_1, x_2]$).

Hidden Layer

The Hidden Layer is a set of Artificial Neurons. Each neuron transforms given input and weight values to an output value through its defined summation function and activation function. In Figure 2.9, the Hidden Layer contains four Artificial Neurons. In most cases, each input neuron from the Input Layer is mapped to each Artificial Neuron in the Hidden Layer to be used for calculation. That is, each Artificial Neuron in the Hidden Layer has its own set of weights and bias (one weight for each input neuron in the Input Layer). In Figure 2.9, there is a total of 12 weights (3 for each Artificial Neuron because there are 3 input neurons), represented by the 12 connecting arrows between the Input Layer and the Hidden Layer. In practice, these weights are represented as a matrix and bias is represented as a vector, all initialized to random values. Matrix multiplication will be performed between the input vector and the weight matrix and the bias will be added (using vector addition) to the result. This matrix multiplication (and vector addition) performs the exact summation process in Figure 2.8 for every Artificial Neuron in the Hidden Layer. For example, in the above network the input vector is of shape 1 x 3 (1 row of 3 input neurons). As stated earlier, we have 4 Artificial Neurons, each of which use each input neuron for its summation calculation. Therefore, we need a matrix of shape 4 x 3 (1 row representing each Artificial Neuron, and 3 weights corresponding to the 3 input neurons). We also have a bias vector of shape 1 x 4 (1 row of 4 bias values, 1 bias value for each Artificial Neuron). In order to perform matrix multiplication, the weight matrix must be transposed to a 3 x 4 matrix. After performing the multiplication, a 1 x 4 vector is produced, which the bias vector is then added to. The resulting vector has the activation function f applied to it to produce a final 1 x 4 vector, where each element in the vector corresponds to the output of 1 of the Artificial Neurons.

Output Layer

The Output Layer produces the final output of the network, using another set of Artificial Neurons. The output of the output neurons can represent different kinds of information, such as: probabilities of different categories, projected cost or earnings, etc. A final calculation, similar to the one that transformed the Input Layer through the Hidden Layer will be performed from the Hidden Layer through the Output Layer. In the network above, the previous output of the Hidden Layer is in a 1 x 4 vector, a 2 x 4 weight matrix will be defined with a 1 x 2 bias vector. The same summation calculation will be performed, producing a 1 x 2 vector. A final activation function will be calculated over this vector. The type of activation function used will depend on the type of data the output represents. For example, for output that represents different categories (for a classification learning task), a softmax activation function is normally used.

As a final note, the model described above only contains a single Hidden Layer. In order for the network to truly be considered "deep", two or more Hidden Layers are needed in sequence. All calculations between Hidden Layers work the same as the calculations performed between the Input Layer and initial Hidden Layer. Subsequent Hidden Layers simply treat the output of the previous Hidden Layer as an Input Layer.

2.1.3 Recurrent Neural Network

Recurrent Neural Networks (RNNs) were implemented to handle sequences of data. This is done by using a "persistent memory" throughout the model's calculations.

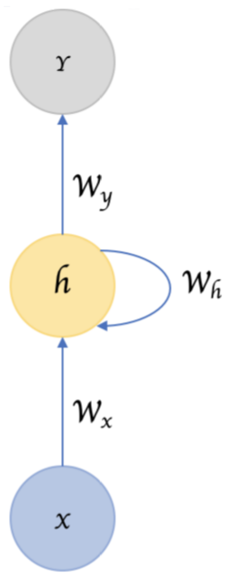


Figure 2.10: Recurrent Neural Network

Recurrence

The model is called recurrent because it performs its calculations in loops (i.e., it recurres), as can be seen in Figure 2.10.

The different node types in RNNs are somewhat similar to the layers of Feed-Forward Neural Networks: “Input” (x), “Hidden” (h), and “Output” (y, or sometimes o). In RNNs, there is instead: “Input Sequence” (x), “RNN Cell” (h), and “Output” (y, or sometimes o). The input sequence is not constrained to being a sequence of single scalar values and can also be a sequence of vectors, matrices, etc.

Essentially, given a sequence of inputs, X, the sequence is broken apart into its individual pieces, called “timesteps” ($X = [x_0, x_1, x_2, ...]$). Each timestep is fed individually to h, the RNN cell, where the input is integrated into the persistent memory that h holds by performing calculations between the current persistent memory state (i.e., the memory from the previous timestep) and the new input. Finally, an output is produced by h at each timestep. Though, depending on the problem being solved, or data being learned, oftentimes only the final output of the sequence is used during loss calculations, learning, classification, etc.

In many cases, RNNs are visualized by “unrolling” the loop for easier understanding. In Fig-

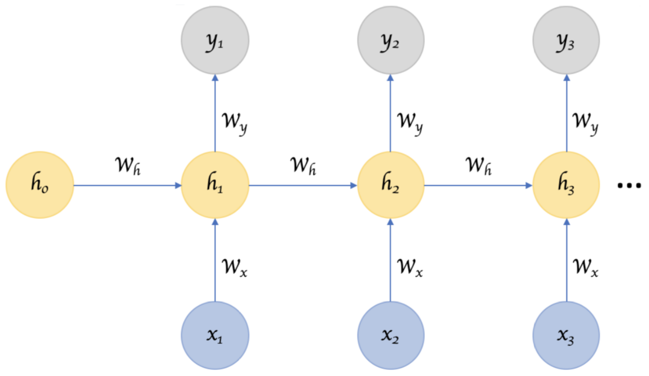


Figure 2.11: Unrolled Recurrent Neural Network

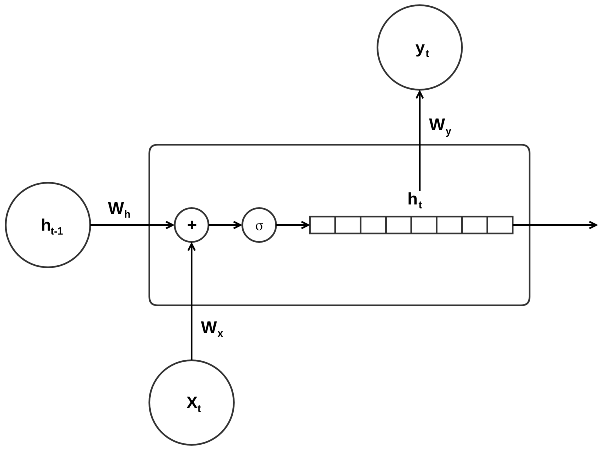


Figure 2.12: Recurrent Neural Network Cell

ure 2.11 we can more easily see the timesteps of X ($X = [x_1, x_2, x_3, ...]$). Further, each h denotes the changing of the persistent memory state it stores. That is, the memory state of h_1 is different from the memory state of h_0, h_2, etc. Each output y is also different at each step (being calculated from the specific state of h at the corresponding timestep).

The Recurrent Neural Network Cell

The RNN Cell holds a vector (or matrix) of some length. This vector is the persistent memory that new input is integrated into and the output is calculated from, as shown in Figure 2.12:

As input, the cell takes x_t (i.e., x at timestep t) which has matrix multiplication performed on it by W_x. This weight matrix performs two purposes. The first is the normal purpose of weighting the different elements of the input. The second is to transform the input shape into the same shape as the persistent memory for proper integration. We also see h_{t-1}, which is the previous state (or values) of the persistent memory of the RNN Cell. Matrix multiplication is performed between it and W_h. The resulting vectors (or matrices) of $x_t W_x$ and $h_{t-1} W_h$ are then added together through vector (or matrix) addition, as shown by the $+$. The result has an activation function performed on it (in Figure **??**, it is the sigmoid, σ, function, but TanH, ReLU, or any other activation function may be used). This results in the new h_t vector (or matrix) for the persistent memory of the RNN Cell for that timestep t. Then h_t is used to produce the output for that timestep, y_t, by performing matrix multiplication between h_t and W_y. Finally, the persistent memory, h_t, is kept to be used for the next timestep, where it will then be used as the new h_{t-1}. This will continue until all input in the sequence X is used.

It is worth noting that in Figure 2.12 h_{t-1} appears to be coming from outside the RNN Cell. This is because we are visualizing the cell in its "unrolled" version. However, this is simply for ease of visualizing the flow of the model. In actuality, h_{t-1} and h_t are the same vector (or matrix) representing the persistent memory of the RNN Cell (represented by the loop in the normal "rolled" RNN model). The only difference between the two is that h_{t-1} is the state of the persistent memory at timestep $t-1$ and h_t is the updated state of the persistent memory at timestep t. They only appear to be separate for ease of understanding. Similarly, the weight matrices (W_x, W_h, and W_y) are also the same matrices between the different timesteps. Again, in the unrolled version of the RNN model they only may appear to be different for ease of visualization and understanding, and can be seen as the same weights in the normal "rolled" RNN model. Further, the shapes of the weight matrices must transform their respective vectors (or matrices) to the proper shapes for calculations and output.

2.2 Word Embeddings

Deep Learning works well for numerical data or data that has a meaningful underlying numerical representations (e.g., pixels in an image). That is, in order to apply deep learning to a problem, a numerical representation of the data is required to perform the necessary calculations. Text,

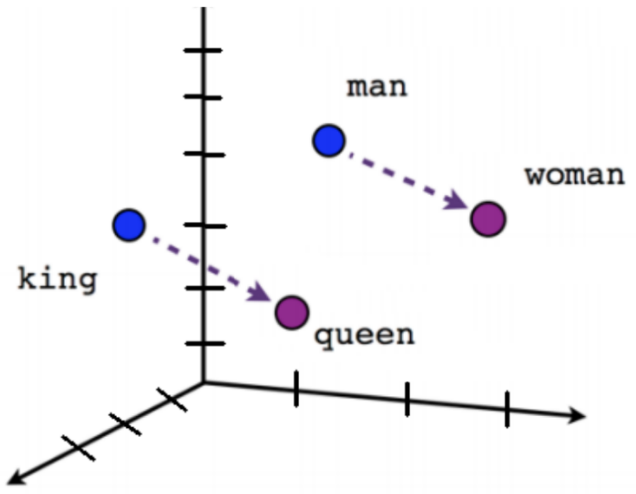


Figure 2.13: Word Embedding Vector Space

however, has no meaningful numerical representation (i.e., though text is represented internally as a numerical encoding in a computer, the numerical assignment is arbitrary and does not represent the semantic meaning of word). However, since a numerical representation is required to apply deep learning to NLP, real-valued vector representations are constructed to represent words, called "word embeddings". Each of the elements in the word embedding represents some semantic feature of the word it defines.

For example, the words *king*, *queen*, *man*, and *woman* may have the word embeddings: *king* = $[2, 0, 3]$, *queen* = $[2, 2, 2]$, *man* = $[0, 1, 4]$, and *woman* = $[0, 3, 3]$. These can be shown in the word embedding vector space, as seen in Figure 2.13.

To determine if we have high quality word embeddings, an exploration of the space will reveal that words with similar semantic meaning (like *king* and *queen* or *man* and *woman*) will cluster together in the vector space. That is, words of similar semantic meaning should have similar vector representations (or word embeddings). We would also hope to see words differentiated by similar ideas or concepts to be separated by roughly the same distance and in the same direction. For example, in Figure 2.13 *king* and *queen* are separated by the same distance and direction as *man* and *woman* because they are differentiated by the same concept of gender.

It should be noted that the examples of *king*, *queen*, *man*, and *woman* use a vector representation (or word embedding) of length three. However, the normal length for these vector representa-

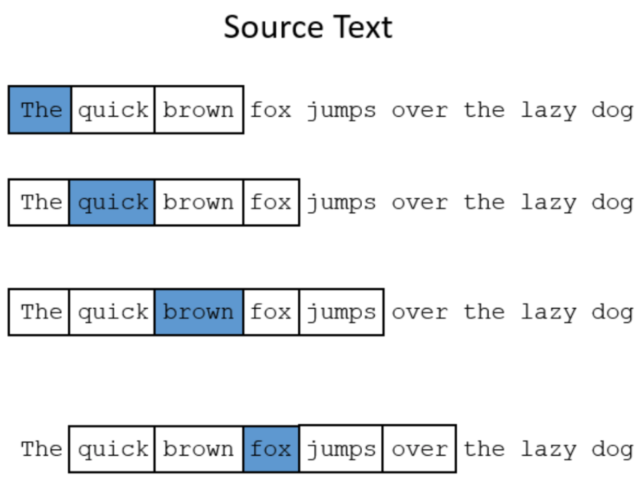

Source Text
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

Figure 2.14: Identifying N-Gram Training Context

tions are between 50 – 300.

To create these word embeddings, different deep learning models and techniques can be used to learn the embeddings using a large corpus of example text. These embeddings are then used for NLP analyses in deep learning models.

2.2.1 N-Gram

N-Gram is a technique used to train word embeddings. It is based on the assumption that the meaning of a word can be derived from its usage in its language. In order to learn a given "target word", "context words" are identified that surround the target word in text examples. These context words and target word are used in a neural network for prediction to learn the embeddings.

For example, in Figure 2.14 an example (*The quick brown fox jumps over the lazy dog*) sentence from a corpus is shown. A word embedding needs to be constructed for each word in the corpus. To begin, every unique word is listed in a vocabulary, and each word in the vocabulary is mapped to a word embedding. To learn the word embeddings, the word embedding for each word in the vocabulary is initialized to random values. Then every word in the corpus is visited from beginning to end, as shown by the sliding blue window in Figure 2.14. Whichever word is in the blue window is the target word for that training instance. A window size is defined to gather context words that surround (before and after) the target word. For example, in Figure 2.14, the window size is 2.

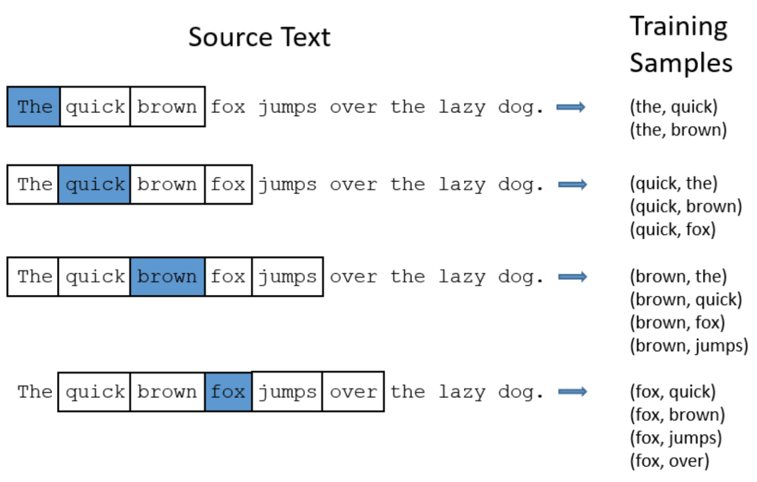


Figure 2.15: Identifying Word2Vec Training Context

Once the context for the target word are identified (for the target word *brown* the context would be *[the, quick, fox, jumps]* and for the target word *fox* the context would be *[quick, brown, jumps, over]*), these are used in a neural network for a learning prediction task.

There are two different approaches for the learning task. Using the target word to predict the context, or using the context words to predict the target. To predict a word, a classification prediction task is performed where each word in the vocabulary is a class. Then, based on the performance of the neural network, the weights of the neural network model as well as the word embeddings themselves are learned. This learning step modifies and trains the embeddings so that the embedding values represent the semantic meaning of each word based on the context it appears in.

2.2.2 Word2Vec

Though the N-Gram model has shown to produce quality embeddings, it is very inefficient. Since a vocabulary can easily contain thousands of words, the classification task that predicts over every word in the vocabulary is attempting to predict over thousands of classes. The softmax operation to predict over so many classes needs a large amount of time to calculate. Further, to obtain high accuracy over such a large classification task requires a large amount of training. The Word2Vec [50, 51] model is used to overcome this inefficiency.

The basic approach is very similar to N-Gram. Given a target word, collect surrounding context words and use those in a neural network to perform a classification prediction task to train word embeddings. As can be seen in Figure 2.15, the identification of context words for a target word is the same. However, instead of all the context words being used in a single training task, the context words are broken up and individually paired with its target word. In Figure 2.15, for example, the target word *fox* is separated into four pairs, one for each of its identified context (*(fox, quick)*, *(fox, brown)*, *(fox, jumps)*, and *(fox, over)*). These training pairs will be used in a neural network to classify if the pairs are related. That is, is the given pair a target with its context or not. Obviously, the current set of pairs are all positive target-context pairs (as we just identified them from the corpus). In order to have negative pairs (a target with a non-context word) random words are chosen from the vocabulary and paired with *fox*, called negative sampling. For example, some negative samples may be *(fox, lazy)*, *(fox, sun)*, *(fox, the)*, and *(fox, apple)* (assuming *sun* and *apple* are also in the vocabulary). All the pairs (positive and negative) are then used in a neural network to predict between only 2 classes, positive or negative (i.e., target with context or target with non-context, respectively). The modification of the word embeddings is performed during training, just like in N-Gram. In this way, we are able to learn the word embeddings with a much more efficient approach.

2.3 Code as a Natural Language

How to analyze natural language has been studied and explored for many decades. Beginning with attempted analyses of language and grammar, moving to statistical models of "utterences" or "occurances", and most recently applying machine learning to replace purely statistical models. Traditionally, static and dynamic analysis has been the state-of-the-art in code analysis and processing. Since code is an "artificial" language with a well-defined grammar and structure, the application of natural language processing to code may seem questionable. However, in the paper *On the Naturalness of Software*, Hindle et al. [32] argues that while code is an artificially constructed language, the behaviour of how developers use code (i.e., its regularity and predictability)

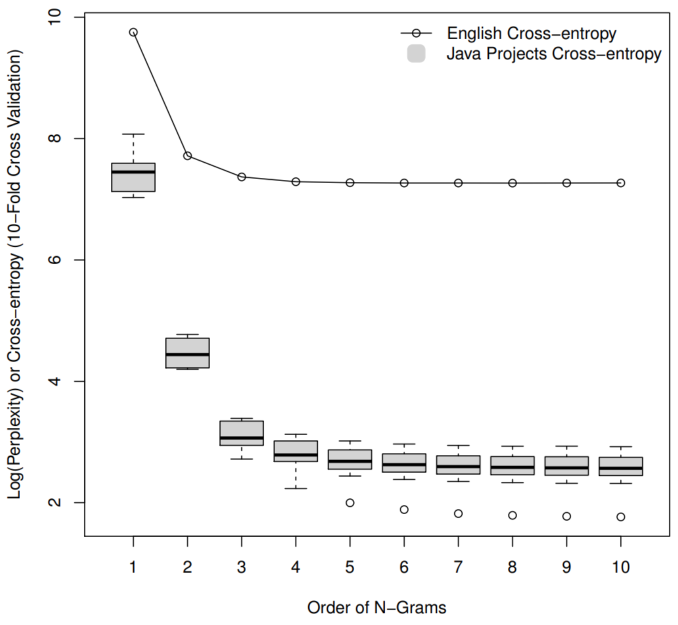


Figure 2.16: English cross-entropy vs. Java Code cross-entropy

are near identical to that of natural language. This similarity makes code a suitable candidate for applying natural language processing for analysis.

2.3.1 On the Naturalness of Software

A basic property of natural language that current techniques of NLP take advantage of is that natural language may be potentially complex, but what is actually written and spoken is mostly regular and predictable. The work by Hindle et al. [32] argues that software is very much the same and so should be able to be modeled by NLP techniques just like natural language: *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks*.

The entropy (or "suprisedness") of natural language and the Java programming language are compared over unigrams to 10-grams, as shown in Figure 2.16. The same trend of entropy can be seen in both the English natural language and the Java programming language.

Further, an NLP code prediction task is performed and integrated into the Eclipse IDE. The

NLP integration is shown to improve the ability for Eclipse to correctly suggest the next code element in a given program.

These experiments are great evidence that while code is artificial, it exhibits a naturalness very similar to natural language. This implies that NLP techniques should be applicable to code processing and analysis.

CHAPTER 3: OVERVIEW

The remainder of this work focuses on solving the static analysis limitations as they relate to privacy compliance verification.

In Chapter 4, the limitations of static analysis are realized through an example project. The project describes a static analysis approach to privacy compliance between an Electronic Medical Record System (EMRS) and The Health Insurance Portability and Accountability Act of 1996 (HIPAA), which are government privacy regulations for health systems, through the application of model checking. The project shows how specifications can be extracted from both the privacy policy and the software system. Model checking is then used to determine compliance using these specifications. That is, all specification actions performed by EMRS related to access and maintenance of privacy data must be allowed by the HIPAA specifications. The project shows how static analysis can successfully perform privacy compliance, but then discusses the difficulties of applying those techniques to health systems at large. This shows the need for generalization and automation of manual tasks of static analysis for privacy compliance verification.

There are three main components to privacy analysis, that is: identification and extraction of privacy specifications in privacy policies, identification and extraction of privacy specifications in software code, and finally the resolution of the two specifications together to determine compliance. The final comparison between specifications is often the simpler component and so this work does not focus heavily on it.

In Chapter 5, the automation of the privacy policy analysis component is explored. This is done through the automatic maintenance of a privacy domain ontology. An ontology is a dictionary of terms or phrases and the relationship between them. In this work, the relationships are hypernymy. That is, what terms or phrases are a higher-order term or phrase. For example, “device identifier” is a hypernym of such terms as: “ip address”, “IMEI”, “android id”, etc. Ontologies are significantly useful for natural language processing as natural language (especially privacy policies) is ambiguous. By maintaining an ontology, that ambiguity can be mitigated by traversing the relationships between terms to determine their relevancy to the problem. This work shows that it is possible to use deep learning natural language techniques to maintain such an ontology to utilize for privacy policy analysis.

In Chapter 6, the challenges of natural language processing and natural language processing on code are discussed and addressed. The concerns of NLP are explored by plotting pre-trained word embeddings of code elements in their vector space. The plots should show clustering of similar code elements. Such clustering would indicate high quality vector representations. However, little clustering is observed. The novel learning approach is then described and then used to pre-train word embeddings of code elements. After plotting the embeddings in their vector space, a significant improvement in clustering is observed.

In Chapter 7, the automation and generalization of the software code analysis component is described. A project is discussed that attempts to predict if, given a stack trace of method calls, a stack trace will access sensitive or non-sensitive privacy data. The novel learning approach is applied and compared to other current, standard deep learning NLP techniques. It shows that there is a significant improvement in the predictive power of the model using the novel learning approach. This work shows it is possible to use deep learning natural language techniques to analyze software code.

In the final Chapters 8 and 9 the thesis statement is evaluated using the results throughout this work and then the work is concluded with a final discussion and future work.

CHAPTER 4: PRIVACY COMPLIANCE WITH STATIC ANALYSIS

Static analysis techniques are the current state-of-the-art in determining privacy compliance. This requires analysis and extraction of models, specifications, or patterns from the natural language privacy policy and the software code. The analysis of the two are then compared to determine if the software code is in compliance with the privacy policy. That is, the software code does not contain any additional privacy-related functionality not dictated within the privacy policy. The following work [39] is a static analysis approach using model checking of privacy compliance between a medical software system and The Health Insurance Portability and Accountability Act of 1996 (HIPAA) which is a government bill that outlines privacy requirements for the collection, storage, access, and maintenance of medical information.

4.1 Verifiable Assume-Guarantee Privacy Specifications for Actor Component Architectures

Many organizations process personal information in the course of normal operations. Improper disclosure of this information can be damaging, so organizations must obey privacy laws and regulations that impose restrictions on its release or risk penalties. Since electronic management of personal information must be held in strict compliance with the law, software systems designed for such purposes must have some guarantee of compliance. To support this, we develop a general methodology for designing and implementing verifiable information systems. This paper develops the design of the History Aware Programming Language (HAPL) [74] into a framework for creating systems that can be mechanically checked against privacy specifications. We apply this framework to create and verify a prototypical Electronic Medical Record System (EMRS) expressed as a set of actor components and first-order linear temporal logic specifications in assume-guarantee form. We then show that the implementation of the EMRS provably enforces a formalized Health Insurance Portability and Accountability Act (HIPAA) [2] policy using a combination of model checking and static analysis techniques.

We aim to develop a full framework with a working prototype of an Electronic Medical Record System (EMRS) that can be statically checked against formal privacy policy specifications for HIPAA. In order to bridge the gap between policy specifications and our implementation, we develop formal system specifications in assume-guarantee form [40]. Static analysis techniques are applied to verify that the implementation matches this specification, while a model checker is used on small specification sets reflecting the formal specifications to ensure that the system's formal specification enforces HIPAA.

4.1.1 Privacy Policy Specification

The privacy policy specification language used [16] decomposes policy formulas into norms by restricting temporal logic formulas to a form similar to that done in the work of Barth et al. on Contextual Integrity [9]. A positive norm allows a message transmission if the condition associated with it holds, while a negative norm allows a message transmission only if its condition holds. An action is thus allowed by the policy if it satisfies at least one of the positive norms and all the negative norms. A notable difference from Contextual Integrity is that our specification language is limited to a restricted subset of first-order temporal logic (FOTL) that makes our privacy policies enforceable.

We use this language to formalize the restrictions and requirements of HIPAA as a set of these communication norms. Of note, the disclosure of Personal Health Information (PHI) is represented by the sending of a message from a principal (i.e., person) that knows information about a subject to another principal that does not. Additionally, principals hold certain roles (e.g., psychiatrist, patient) and messages are sent for specific purposes (e.g., treatment, billing).

A communication action is denoted by *send(p, q, m)*, in which p is the sending principal, q is the receiving principal, and m is the message being sent. Each message contains a set of principal attribute pairs. The predicate *contains(m, q, t)* holds if message m contains attribute t of subject principal q, such that the recipient of message m would learn the attribute t about q. Roles can be bound to principals via the *inrole* predicate, where *inrole(p, r)* holds if *(p, r)* $\in$ *roleAssignment*.

Similarly, *for-purpose(m, u)* enforces that message m is sent for purpose u.

4.1.2 System Framework and Specifications

In the actor model [3, 31], a software system is considered to be a collection of concurrently operating actors that communicate through asynchronous message passing. Each actor has a mailbox through which it receives messages, and an actor may only act in response to these messages. Based on its state/behavior, an actor reacts to the messages it receives one at a time (but not necessarily in the order they arrive) by performing some action. The synchronous actions an actor takes in response to the receipt of an asynchronous message can involve sending a finite number of messages to other actors it knows about, creating a finite number of new actors, or changing its state/behavior so that it will take different actions in response to future messages.

Bengt Johnsson et al. [40] established the use of linear temporal logic (LTL) as sets of assume-guarantee specifications for formally specifying behavior of an open system. In short, the way in which a system interacts with its environment can be specified through assumptions on the behavior of its environment and guarantees on the behavior the system can, or will, exhibit if the assumptions on its environment hold true. Safety assumptions and safety guarantees describe that bad things cannot happen, by stating conditions that must hold on the information the system receives from its environment and the information the system sends to its environment, respectively. Liveness guarantees describe that good things must eventually happen, by stating conditions that force the system to eventually communicate information to its environment. Assume-guarantee specifications are well-suited for formally specifying the behavior of an actor system by describing under what conditions actors can send or receive certain types of messages

We have developed a framework for implementing actor systems that meet assume-guarantee specifications and that can present a compliant system to an end-user (e.g., an auditor) of that system [45]. Broadly, the framework shown in Figure 4.1 takes in formal privacy regulations and organizational policies that are manually refined into assume-guarantee specifications of the information system. This framework is based on and implements our HAPL language. This lan-

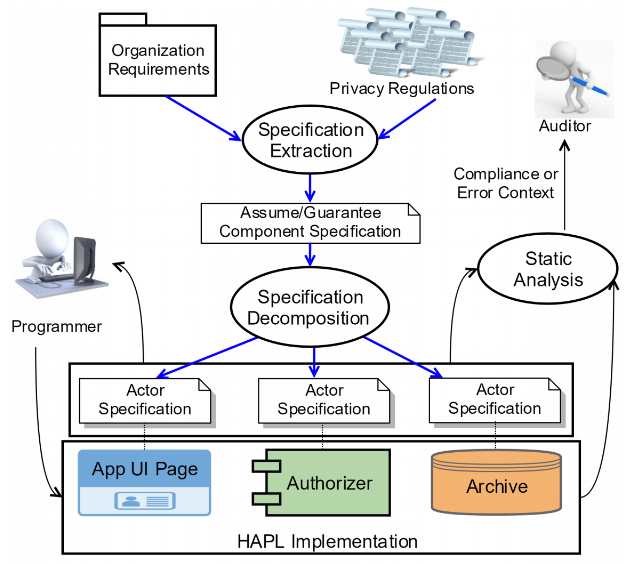


Figure 4.1: Framework

guage can be used to define applications with attendant web-based user interfaces that comply with specification. This compliance is guaranteed by a static analysis phase requiring that the HAPL implementation honors the specification. Failing that, the static analysis phase will return with error contexts explaining the failure. Finally, we evaluate the framework by implementing a functioning EMRS prototype that complies with a representative subset of HIPAA requirements.

HAPL is a general purpose, actor-based, imperative programming language that incorporates history queries. HAPL is split into a user interface specification [46], actor behavior specification, static analysis and interpretation. The language allows for defining actors and the principals they represent, as well the role in which actors represent principals. In HAPL, actors are considered to be acting in a particular role for a particular principal whenever they send or receive a message. This assumption simplifies our approach to our assume-guarantee specification and decomposition.

As per the actor paradigm, computation in HAPL is done through creation of and communication between actors. We constrain HAPL actors to synchronous control only of private data, and communication with other actors only by sending asynchronous messages.

The EMRS is decomposed into a set of communicating actors (as shown in Figure 4.1) whose behavior is specified as a set of FOTL formulas in assume-guarantee form. Once we have our EMRS specifications in place, we would like to verify that the system behaves correctly by showing that it is HIPAA-compliant. We have a set of FOTL formulas that describe norms of communication in terms of when PHI can be disclosed according to HIPAA. Thus, we can verify HIPAA-compliance of our specifications by showing that any disclosure by our system entails a valid disclosure described in the policy norms.

We use static analysis to bind HAPL code to the constraints detailed in our assume-guarantee specifications. By making statements about type-correctness in assignment, message parameters, message handlers, and send message inroles, as well as verifying that certain properties hold in the abstract syntax tree, we are able to verify compliance with certain assume-guarantee specifications. The static analysis phase therefore reports non-compliance and fails compilation so that an actor system is not generated where it cannot be proved that the code matches appropriate assume-guarantee specifications.

To demonstrate the language and evaluate our framework, we build a EMRS prototype with simple implementations of five use cases specified to comply with HIPAA. These are intended to be a representative subset of the release scenarios described in HIPAA. In particular, this includes cases involving releases that are required, releases that are permitted without consent, and releases that are permitted only with authorization. Those five use cases are:

- Record contains no PHI: If a message contains no PHI, then that information may be released without authorization.

- Patient requests own PHI: HIPAA has a liveness requirement that any patient request for their own PHI records must eventually be fulfilled. Thus any request by a patient role for their own records must be honored by the EMRS.

- Originating doctor requests patient's psychotherapy note PHI: A doctor who is acting as a physician for a patient may obtain that patient's psychotherapy notes without the consent of any principal if that physician is also the author of that PHI.

- Doctor requests patient's non-psychotherapy PHI: A doctor who is acting as the physician for a patient may obtain that patient's PHI without any principal's consent, so the prototype

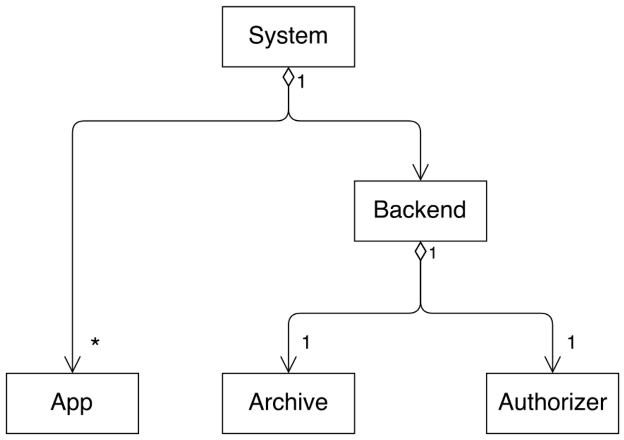


Figure 4.2: EMRS Decomposition

will honor requests by a doctor for patient data.

- Third party requests patient PHI: Third parties may request patient PHI for marketing purposes. These requests may be honored if it is determined that the patient that is the subject of this PHI has authorized release of it for this purpose.

As a first step towards creating specifications for the HAPL EMRS, we conceptualized a decomposition of the system into a set of actors, each of which has a set of responsibilities that it upholds by communicating with the other actors. As shown in Figure 4.2, we decompose the HAPL EMRS into three actor types: the App, the Archive, and the Authorizer.

There are many App actors which act on behalf of human principals and provide them an interface to access the EMRS. Through this interface, a principal may request that the EMRS release medical records to them and receive the corresponding reply. An App also allows patient principals to grant authorization for their records to be released to another principal by sending a message to the Authorizer, or reject an incoming request from the Authorizer for such an authorization. Additionally, an App allows doctor principals to add records to the EMRS by sending them to the Archive.

The Archive and Authorizer are both singleton actors which act on behalf of the organization whose combined behavior can be composed into a component referred to as the Backend. The role of the Archive is to store and retrieve records about patients as requested, and the Authorizer keeps track of information necessary to make policy decisions for the EMRS and makes those decisions for the Archive when requested. When an App sends messages which request some action on a record to be performed, the Archive forwards the necessary information to the Authorizer. The Authorizer then attempts to make the policy decision according to the actor specification and previous system actions. If the policy decision cannot be concretely made from the system history, the Authorizer sends an authorization request message to the App belonging to the subject of the record. After receiving the final decision back from the App if necessary, the Authorizer returns the appropriate response message to the Archive, and the Archive sends the corresponding final reply to the App which made the original request.

Actor specifications are the foundation on which higher-level specifications (which have a similar structure) are established. Each actor type has an assume-guarantee specification that prescribes how instances of that actor type may interact with its environment (i.e., the messages it may or must send to other actors).

4.1.3 Model Checking and Evaluation

We are able to apply model checking techniques to evaluate whether the assume-guarantee specifications entail the privacy policy in the EMRS prototype. Automation of the proof process provides a level of confidence in the validity of the EMRS prototype that is essentially too tedious, or even infeasible, to achieve by manual means.

We use model checking to verify that the EMRS specifications imply all negative norms and at least one positive norm of the privacy policy before any disclosure of PHI. We selected the symbolic model checker NuSMV [17] to perform the checking because the size of the specification formulas can be large, causing the state space of the model to be large. Additionally, NuSMV supports both past and future temporal logic operators, which are used in both the formal policy norms and EMRS specifications.

NuSMV is a symbolic model checking tool for verifying a model of a finite-state system against properties specified in temporal logic. The model is specified by states and state transitions. A state is an assignment of all variables within the system. A state transition defines an allowable change in variable assignments in response to an event or condition.

We encode the terms of the FOTL EMRS using LTL and policy specifications in NuSMV. NuSMV then symbolically determines if the formulas are verified, and if a violation is found, a counterexample is generated.

We wish to show that the EMRS satisfies the privacy policy. To do this, the EMRS specifications must entail all of the negative norms and at least one positive norm of the privacy policy for each disclosure of PHI. To show this, we encode EMRS entailment formulas, in LTL form, in NuSMV. All entered formulas were satisfied by the EMRS. This means that the EMRS specifications implied at least one positive and all negative norms, thereby showing that EMRS entailed the privacy policy.

4.1.4 Limitations

As described, the static analysis approach solves the privacy compliance problem and can even be proved that it will successfully enforce compliance. However, there are some limitations to this approach. The major limitations lie in the lack of automation and generalizability.

In order to extract the specifications from the EMRS system, it must be implemented using HAPL. This language is new and so not only is there a high learning curve to using it, but also all existing systems would have to be re-implemented in HAPL is be analyzed. It is infeasible to expect old systems to re-implemented as well as all new systems to be implemented in HAPL.

All HIPAA specifications must be manually extracted from the policy. This may be acceptable for HIPAA specifically, as the extraction only has to be performed once to apply to all medical systems. If HIPAA is changed or updated, then the manual extraction will have to be redone, but this does not happen often. However, it is not uncommon for business and companies (even hospitals) to also have their own privacy policies in addition to government regulations for their systems. Since each of these privacy policies are unique to the business or company, the manual extraction of privacy policy specifications by a third-party entity (e.g., an auditor, a user, etc.) for all privacy policies is infeasible in real world applications.

Model checking suffers from the state explosion problem. While in the case of HAPL, specifications can be extracted directly from the system (and therefore system states may not need to be traversed) the state explosion problem does not apply. However, if we perform model checking on a system that does not use HAPL, then the system will need to be properly modeled by the model checking tool and states will need to be traversed to determine compliance which will be subject to the state explosion problem.

The specifications must be manually coded into the model checking tool. This means each specification must be inspected for proper variable type declaration (e.g., a free/environmental variable or a dependant variable) and for other variables that may need to be added because they are implicitly assumed by the system or policy, but not by the model checker. The manually coding of a system that may have many dozen or even hundreds of specifications is infeasible in real world applications.

CHAPTER 5: AUTOMATION OF PRIVACY POLICY ANALYSIS

To help solve the limitations of privacy compliance using static analysis, the following work utilizes deep learning to maintain an information-type ontology. This ontology can then be used to identify the types of privacy data collected and processed by apps, as dictated by a privacy policy.

5.1 Learning Ontologies from Natural Language Policies

Several state laws and app markets, such as Google Play, require the disclosure of app data practices to users. These data practices constitute critical privacy requirements, because they often underpin the app's functionality while describing how sensitive, personal data is collected, used, and with whom the data is shared. The requirements that appear in privacy policies mainly focus on abstract information types when describing data practices. Apart from abstraction, various stakeholders can use different words to describe the same domain concept. Abstraction and variability in concept representation are important causes of ambiguity in natural language and they reduce shared understanding among app developers and policy writers. To overcome this obstacle, we propose an automated approach to infer semantic relations among information types and construct an ontology that can be used to guide requirements authors in the selection of the most appropriate information type terms. Our solution utilizes a deep learning model based on word embeddings and Convolutional Neural Networks (CNN) to classify information type pairs as either hypernymy, synonymy, or unrelated. We evaluate our method on a manually-built ontology, yielding predictions that classify the unrelated information type pairs with 0.980 precision and 0.984 recall, suggesting a large reduction in the efforts required for ontology construction.

Mobile apps collect different categories of personal information, such as contact information, photos, and real-time location. This information collection puts sensitive user information at risk since it becomes up to the app developers to protect it. To avoid such mishandlings and maintain the security of such information, app developers often keep strict data requirements over how the data is to be used and collected. These requirements are commonly communicated to users in the form of privacy policies, which inform stakeholders about what kinds of personal information is collected, how the data is used, and with whom the data is shared. Privacy policies are also required by U.S. and E.U. laws.

To ensure data transparency and compliance, methods have been proposed to analyze privacy policies and data practices. For example, Breaux et al. formalized data requirements in privacy policies using Description Logic [13], which can then be used to automatically detect conflicting requirements [14] and to trace data flows across policies of interacting services. Tracing privacy requirements across policies can enhance developers' understanding of third-parties' data practices and comply with legal requirements. Furthermore, others have proposed techniques to trace data requirements from privacy policies to app code using lookup tables, platform permissions, and information flow analysis [62, 83]. These methods depend on a correct lexicon of information types that is generally compiled manually. These information types include phrases, such as device ID, IP address, and location. Moreover, the phrases can overlap in meaning leading to false requirements formalization, e.g., the phrase “WiFi signal strength” can be construed to be a type of location information. Ambiguities commonly found in privacy policies are that of hypernymy, which occurs when a more abstract information type is used instead of a more specific information type (e.g., “device information” instead of “IP address”) [11]. Hypernymy permits multiple interpretations of words and phrases, which leads to inconsistency in traceability. Such interpretations enable the construction of a reusable ontology that illustrates information types and their semantic relationships via hypernymy.

Ontologies are particularly useful in requirements when dealing with abstractions in human language. For example, “operating system,” hardware model,” “application type,” “application version,” “browser type,” “browser version,” and “unique identifiers” are all a kind of "device identifier." However, such relationships are not always so clear especially when writing or reading requirements or policy policies. By deferring to a pre-constructed knowledge graph which shows known relationships between phrases used in the domain (i.e., an ontology), these relationships can be applied. This allows for more precise requirements to be inferred from abstract language.

The current state of the art for ontology building requires manual, human comparisons of each node. Not only is this tedious, but it is susceptible to human error due to fatigue and potential lack of domain knowledge from the analysts. Furthermore, the language used may evolve, requiring continuous rebuilding of the ontologies making manual generation less viable. For these reasons, analysts may be hesitant in using ontologies though they have been empirically shown to be useful [66].

To address these issues and enable easier, more consistent ontology construction, we developed an empirical method to learn and construct a formal ontology from a naive set of all pairs of information types contained in a lexicon.

Relation Classification Model

Prior work to construct a privacy ontology [34] requires comparing information types with every other type in a lexicon and assigning a semantic relationship to each pair using seven heuristics. The required effort for this task is quadratic , where is the number of information types in the lexicon, and is the amount of time required to assign a relationship to a pair, estimated at 20 seconds [33]. In addition, a new policy introduces between 11-36 new types that are not encountered in the existing lexicon [11]. Considering app markets contain hundreds of thousands of apps that change daily, we need to automate ontology construction. Thus, we propose to predict candidate relationships between information type pairs. Unlike syntactic processing (e.g. part-of-speech), our approach employs a convolutional neural network (CNN).

Figure 5.1 shows the learning architecture for the relation classification model with a pair of information types as input. Throughout the paper, we present the information type pair as (information-type$_{LHS}$, information-type$_{RHS}$), e.g., (device information, device ID), where LHS (left-hand side) and RHS (right-hand-side) indicate two predicates in an asymmetric ontological relationship. The input information types in a privacy policy lexicon can be from a single statement in a policy, different sections of a single policy, or completely different policies. Given an information type pair, the Embedding layer first maps the words in an information type to their cor-

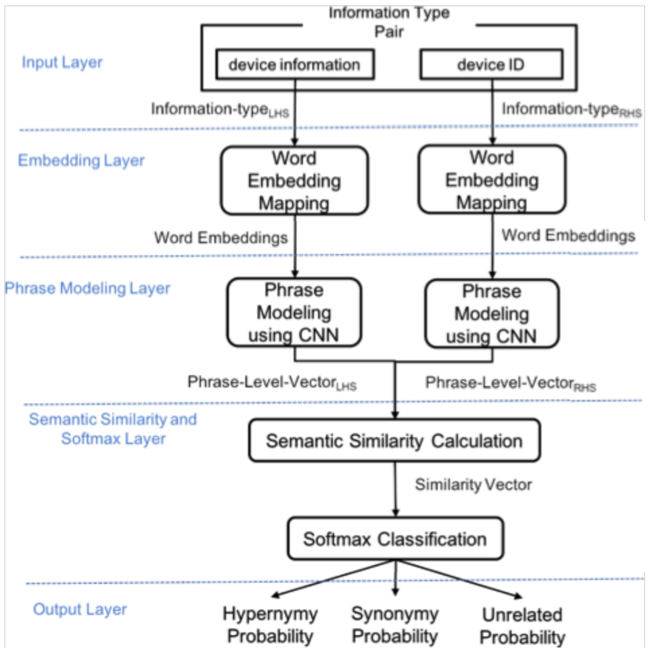


Figure 5.1: Ontology Learning Architecture

responding word embedding vectors. Second, word embeddings are fed into the Phrase Modeling layer, creating a phrase-level semantic vector for each information type phrase. Third, the Semantic Similarity Calculation compares the direction and distance of the two phrase-level vectors and generates a similarity vector. Finally, the similarity vector is input to Softmax, generating three probabilities corresponding to hypernymy, synonymy, and unrelated. We select the most probable relation for each information type pair.

For the word embeddings layer, each word in an information type phrase is presented using a pre-trained 200-dimensional vector called a word embedding [10]. To create domain-specific word embeddings, we followed the approach by Harkous et al. [25] and trained the Word2Vec model using 77,556 English privacy policies collected from mobile applications on the Google Play Store.

In the phrase modeling layer, as seen in Figure 5.2 we transform word embeddings of an input information type to a low dimensional, fixed-sized vector using CNN with three different filter widths [72]. Implementing CNN with multiple filter widths captures local semantics of n-gram of various granularities [71]. In our case, convolutional filters with widths 1, 2, and 3 capture the

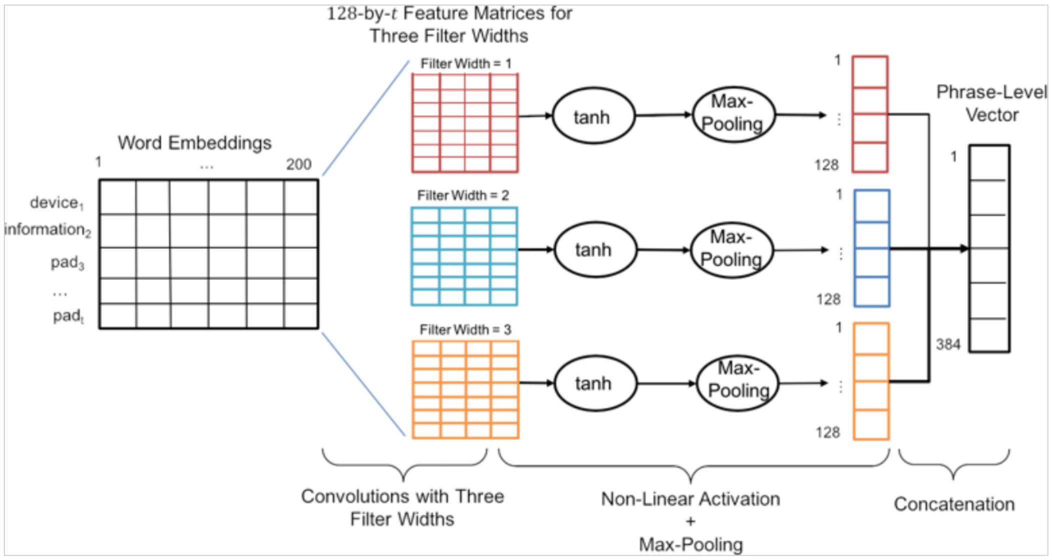


Figure 5.2: Phrase Modeling Layer

semantics of unigrams, bigrams, and trigram, respectively.

The semantic similarity and softmax layer compares the input phrase-level vectors from the Phrase Modeling layer. We adopt the structure proposed by Tai et al. [70], where the direction and distance of the two input vectors are compared using the following equations. Two vectors PLV_{LHS} and PLV_{RHS} refer to Phrase-Level-Vector$_{LHS}$ and Phrase-Level-Vector$_{RHS}$ in Figure 5.1, which are shortened for simplicity in the equations.

$$dir = PLV_{LHS} \bigodot PLV_{RHS} \tag{5.1}$$

$$dis = |PLV_{LHS} - PLV_{RHS}| \tag{5.2}$$

$$sim = \sigma(W * dir + U * dis + b) \tag{5.3}$$

$$P_{relation} = softmax(W_p * sim + b_p) \tag{5.4}$$

Equation 5.1 compares the direction of two semantic vectors and for each dimension using the point-wise multiplication operator. For calculating the distance between PLV_{LHS} and PLV_{RHS}, we utilize the absolute vector subtraction presented in Equation 5.2. To integrate the results of Equation 5.1 and Equation 5.2 on PLV_{LHS} and PLV_{RHS} , we use a hidden sigmoid layer pre-

Table 5.1: Experiment 1 Results

	Direct Hypernymy	Synonymy	Unrelated
Precision	0.483	Undefined	0.991
Recall	0.775	0	0.981
F1 Score	0.595	Undefined	0.985

sented in Equation 5.3. The similarity vector as the output of the function is then sent to a Softmax classifier as shown in Equation 5.4 to predict the probabilities of hypernymy, synonymy, and unrelated. We select the prediction with the highest probability as the relationship between information-type$_{LHS}$ and information-type$_{RHS}$.

Evaluation

As our ground-truth, we utilize an ontology that is manually built from a privacy policy lexicon. This lexicon was extracted from 50 privacy policies [66]. It contains phrases that correspond to platform information types, defined as "any information that the app or another party accesses through the mobile platform that is not unique to the app." Each information type has a frequency, or number of times the type appeared in annotations of the 50 policies. Hosseini et al. manually constructed the platform ontology from the lexicon by applying seven heuristics that were identified through grounded analysis of five privacy policies [34]. The platform ontology contains 367 information types, which are used to comprise 1,583 hypernymy and 310 synonymy relationships between pairs of information types.

Two experiments were constructed from the platform ontology using a 70% to 30% training to testing split. In experiment 1, we classify whether a new information type pair describes a direct hypernymy, synonymy, or unrelated (i.e., otherwise) relationship. In experiment 2, we classify whether a new information type pair describes a transitive (or indirect) hypernymy, synonymy, or unrelated (i.e., otherwise) relationship. The results for experiment 1 are in Table 5.1 and the results for experiment 2 are in Table 5.2.

Table 5.2: Experiment 2 Results

	Transitive & Direct Hypernymy	Synonymy	Unrelated
Precision	0.856	Undefined	0.980
Recall	0.862	0	0.984
F1 Score	0.858	Undefined	0.981

Discussion and Future Work

We now discuss our results, limitations, and improvement required to address some of the challenges in our work. Further, we discuss the impact of our work on creating a shared understating between privacy authors, developers, and regulators.

Regulators and app markets require app developers to describe their data practices in privacy policies. Apart from abstraction, stakeholders can use different words to describe the same data concept. Using semantic relations, such as hypernymy and synonymy, one can formalize relationships between concepts to create a shared understanding. The model proposed herein identifies hypernymy relationships between information type pairs with 0.858 F1 score. The model also identifies unrelated information type pairs with 0.981 F1 score, greatly reducing the search space of candidate relationships.

We now list the challenges and limitations of our work. Based on the results in Experiments 1 and 2, the model failed to predict synonymy relationships due to insufficient training examples. Despite biasing weights for prediction, which enhances hypernymy predictions, synonymy predictions remain unaffected. To reduce the effect of minority class, over-sampling methods, such as random over-sampling and Synthetic Minority Over-Sampling Technique (SMOTE) [15], can be used. Random over-sampling simply repeats training examples to reduce imbalance. In contrast, SMOTE generates synthetic training examples based on distance functions.

In Experiment 2, our model misclassifies 972 information type hypernymy pairs as unrelated. One approach to reduce this misprediction could be to use the syntax-based method by Hosseini et al. that infers semantic relations among types that share at least one word [33]. To this end, we utilized their ontology1 constructed upon lexicon (see Section IV.B) to investigate if the misclassified hypernymies can be inferred. Based on the results, we can only infer $56/972$ of hypernyms. Alternatively, we may explore the use of knowledge about siblings and shared ancestors to identify pairs misclassified as unrelated. Finally, the manually constructed ontology may include omissions, since manual classification is subject to fatigue and recency effects, where the analysts recalls the information types that they classified last [34, 55]. We propose to add additional forms of evaluation using preference theory [44] to elicit the relationships among predicted classifications. The results of a preference survey can illuminate the extent of relatedness among pairs and evaluate a sample of potential misclassifications.

Finally, the model is trained on platform information types, which does not include domain-specific information types, such as health-, finance-, dating-, and shopping-related app data, to name a few.

CHAPTER 6: LEARNING MODEL FOR CODE ELEMENT VECTOR REPRESENTATIONS

6.1 Vector Representations Using Current Natural Language Processing

Since we can treat code like a natural language there exists research that has attempted to apply natural language processing (NLP) techniques to code. One of the most successful works [24] used Word2Vec [52] to pre-train vector representations for code elements of Debian and Github code, identifying over about 1 million methods as a dataset. These vector representations are used in a CNN model to predict if a given method is buggy (i.e., contains some bug) or clean. The dataset was split into 80%, 10%, and 10% for training, validation, and testing, respectfully. The prediction on the testing set achieved a precision-recall AOC of .49.

Figure 6.1 plots the pre-trained vector representations (or embeddings) of the code elements in their vector space. Exploring the vector space, we should see similar code elements cluster together. Some general clustering can be identified (e.g., digits, some operators and symbols, and some keywords), but it does not represent the quality of word embeddings desired for NLP deep learning tasks.

In some preliminary experiments [29], we also used a Word2Vec model to learn vector representations for code elements of the JDK8 source code. As shown in Figure 6.2, the pre-trained embeddings also do not cluster well, showing they are not quality vector representations.

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

In this work [27] we describe our first step in applying learning techniques to access control, which is determining good word embeddings to perform learning tasks over. We also discuss the future work on identifying access-control enforcement code and checking access-control policy violations, which are enabled by the word embeddings.

Similar to privacy policy compliance, the ability to verify access control policies and properties

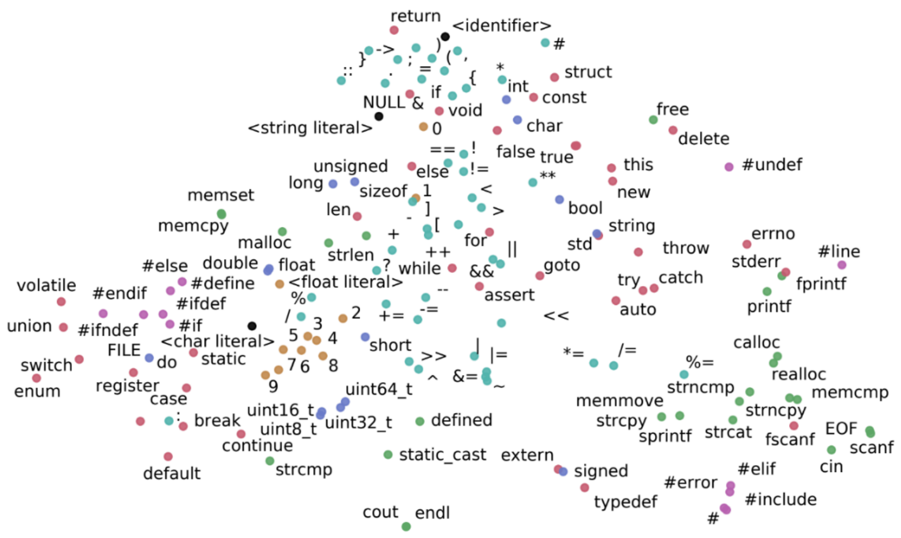


Figure 6.1: Word2Vec Code Element Embeddings

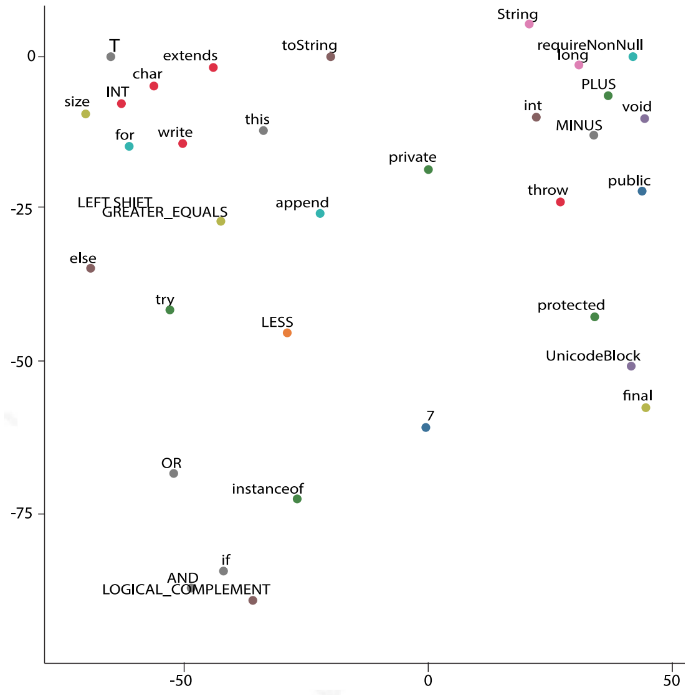


Figure 6.2: Vector Representations for the 35 Most Common JDK8 Code Elements

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 an access control model that defines the access control behavior of the system. 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 a model, 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 in that particular system or systems 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.

We modify the construction of context of the normal Word2Vec model for code elements so that quality word embeddings may be produced. We run the modified Word2Vec model on the access control library Apache Shiro, and evaluate the resulting embeddings by plotting the 99 most common code elements in the library.

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 important in determining how code elements are related to each other, as shown by Peng et al. [60]. Therefore, to help capture the inherent structure of code we will determine code element context using a code's abstract syntax tree (AST). An abstract syntax tree is a tree representation of source code that models its abstract syntactic structure. The types of nodes and edges in the AST are consistent with the grammar of the language of the source code the AST models. An AST will also provide a data structure to easily walk through all code elements in the source code. We use the JavaParser tool [67] 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 its child nodes would provide the context for that node, similar to the work done by Tai et al. [70]. However, the word embeddings generated by this approach were of poor quality. We determined there are a number of issues with this approach but two main problems stand out. One being 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 other is that certain children do not actually affect the meaning of their parent, such as the example given earlier for the *while* loop where statements inside the loop (which are children of the *while* node in the AST) do not determine the meaning of the *while* keyword. We also determined that other previous works that attempted 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. This contributes to the degradation of the quality of the word embeddings which will negatively impact any learning task the embeddings are used for.

We propose to instead 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. Such types of nodes includes, but is not limited to, nodes for: comments, literals, loop statements, method declarations equation operators, etc. We constructed rules for node types by posing the question "What other node types or code elements are needed to determine the behavior of this node type?" For example, a method declaration node is identified by modifiers, parameters, return type, and method body to perform its programmed task; the *while* statement node is identified its conditional expression and any statements that are present that directly affect the behavior of a loop (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 6.1 that details what code elements were included as context for each node type. In the table, the columns 1–3 present the type of AST node, identifier of the AST node, and code elements being used as the 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.

With the rule sets we have created for each node type we have trained word embeddings for the access control library Apache Shiro. Figure 6.3 plots the embeddings for the 99 most common code elements in Apache Shiro.

These code element embeddings do show some improvement for code elements linked to certain node types AST node types (e.g., method declarations), but they are not as drastic an improvement in quality from the other Word2Vec models as desired. This shows that the type of context identified for word embedding training, but we will need to go beyond just modifying context identification and modify the learning model itself.

6.2 Semantic Vector Representations Using Code Element Definitions

The attempts to treat software code like other normal natural languages and apply natural language deep learning techniques directly as before have been met with limited success, performing below current state-of-the-art NLP standards. As discussed in Section 1.2, there are not only many chal-

Table 6.1: Node Type Identifiers and their associated Context

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 Statement	"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

Table 6.1: Continued

Node Types	Identifier	Code Elements Used As Context
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

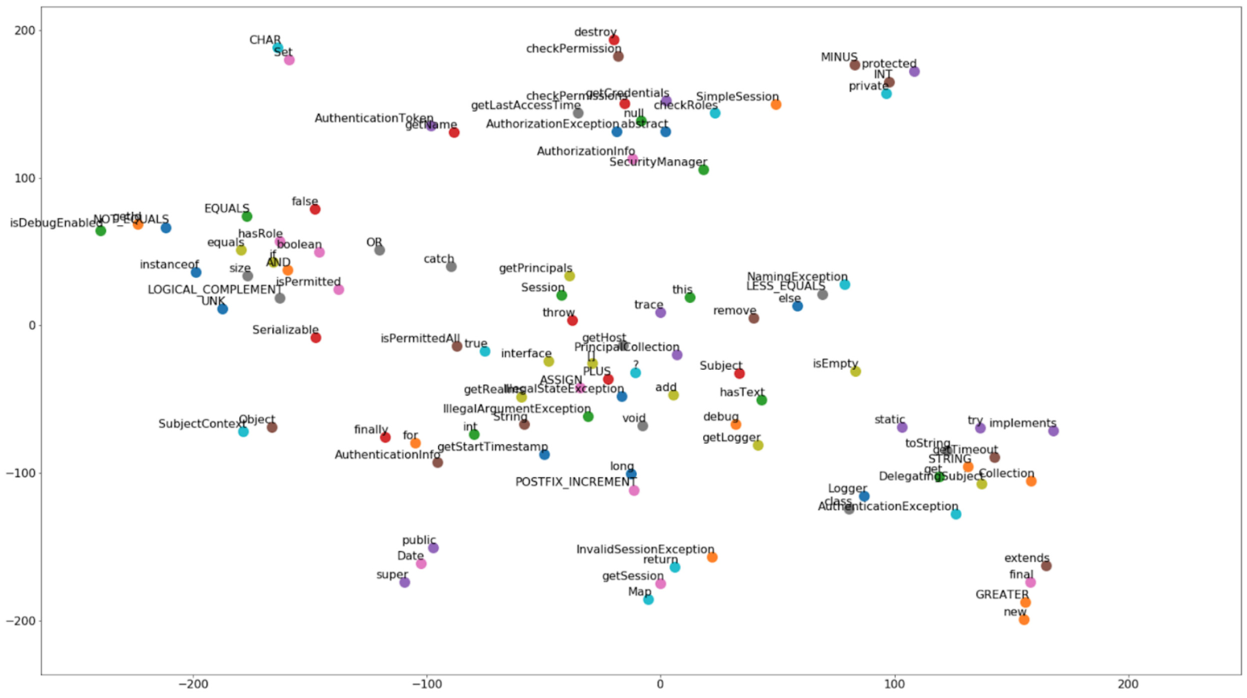


Figure 6.3: Vector Representations for the 99 Most Common Apache Shiro Code Elements

lenges to normal NLP learning tasks, but code has additional challenges that must be addressed. Not only is the syntax of code much more complex than natural language, but the data sparsity problem is far more severe and encountering new code elements happens far more often. While normal NLP is able to mitigate the data sparsity problem and encountering new words by increasing the size of its corpus to include more examples of sparsely occurring words and more words that have yet to be trained, we do not have this luxury in code. That is, other than reserved words and symbols of a programming language as well as standard and common libraries, a system or program is quite unique. A system or program contains many declared and defined code elements (e.g., classes, methods, variables, etc.) that do not occur in any other system or program. Even if the name of a defined code element is common to more than one system or program, it is unlikely that the associated definition or expression is the same. Therefore, we are unable to mitigate the data sparsity problem or encountering new code elements by simply increasing the corpus size. Once an entire program is included in the training corpus, there are no more examples of the unique code elements in that program that can be obtained from anywhere else. Further, regardless of the number of systems or programs included in the corpus, when analyzing a new system or program it is inevitable that many new unique code elements will be encountered.

A novel approach to learning code (and possibly NLP in general) is needed to overcome the unique challenges presented by the nature of code. Instead of only utilizing the classic definition of a language model (i.e., a word's meaning can be defined by its probability of occurrence), this work proposes to also take advantage of the equivalence relationship between a code element and its declaration or definition.

This will allow learning a vector representation for a code element on-the-fly as long as a definition or declaration can be provided for it, and will allow for the learning of new code elements with limited or no pre-training allowing the model to handle new code elements never seen before. Data sparsity is also mitigated as only a definition or declaration need be provided to learn an embedding, meaning only the single occurrence of the code element is required instead of the many occurrences needed in other NLP techniques. Also, in being able to embed a code element directly, the process of learning embeddings (normally only done in pre-training) can be done as a part of the learning task, exposing more information to the learning model, potentially allowing for greater learning and predicting power.

6.2.1 Pre-Processing

Like normal natural language, textual pre-processing and prepartion of software code is needed before it can be used as input to the learning model. This pre-processing is done by utilizing the AST of software code. After the AST is built, it only requires we walk the tree and process each node type as needed. To do this pre-processing, an IntelliJ [38] Plugin was created. IntelliJ was chosen because it is open source, a popular IDE, and it contains a sophisticated API for use in pre-processing code. The code was pre-processed in the following major ways:

- Every code element was delineated by white space for easy tokenization by the learning model program. This separated all keywords, operators (excluding the dot (.) operator), syntax symbols, digits and characters (so only a small, finite set of representative dictionary entries would be needed), and variable, method, and class names.

- Each class, method, and field name is fully qualified. This allows code elements with the same identifier (or simple) name to be entered into the vocabulary without name collision between different classes. It also allows the specific method declaration and body for that name to be utilized when being embedded.

- Parameters of methods are removed from the declaration and added to the beginning of the method body as newly declared local variables. A unique *caller_defined* token is identified to denote that it is a parameter whose value is determined by a calling function. Since parameters are referenced in the method body as normal local variables, it is important that they be included as part of the method body for the learning model to better identify patterns and features.

- The declaration and definition of each class, method, and variable are recorded and grouped into appropriate categories. These will then be printed to a file for ease of input formatting and learning. For classes, the modifiers, field variable member names, method member names, and inner class member names are each grouped together and associated with that specific fully qualified class name. For methods, the modifiers, return type, and method body are grouped together and associated with that specific fully qualified method name. For variables, the modifiers, type, and initialization expression are grouped together and associated with that specific fully qualified variable name.

6.2.2 Learning Model

This work is focused on the Java programming language as it is very popular and covers a wide range of devices and software systems. In the Java language, new code elements that can be declared or defined are classes, methods, field variables, and local variable. These code elements will be the focus of the novel embedding learning approach. Other code elements with no explicit declaration or definition (e.g., keywords, operators, syntax symbols, etc.) will not be learnable in the way described and will be learned implicitly as part of the other code elements' learning, similar to normal NLP techniques. However, most of such code elements are very common throughout code and so should not be sparse and therefore fairly easy to learn.

The new learning model is generalized in Figure 6.4. Essentially, after pre-processing all code to be used for learning and the dataset for the learning task is formatted, a vocabulary and initial embeddings are constructed. For each code element that has no explicit declaration or definition, the code element is embedding normally using the vocabulary. For each defined or declared code element, an embedding is produced for it on-the-fly using its declaration or definition. The intended learning task is then performed on the newly embedded code as would normally be done, producing a prediction for the learning task.

Variable Model

The Variable Model, Figure 6.5, (used for both field variables and local variables) is fairly straightforward. To embed the name of a variable, its expression and modifiers are collected. The expres-

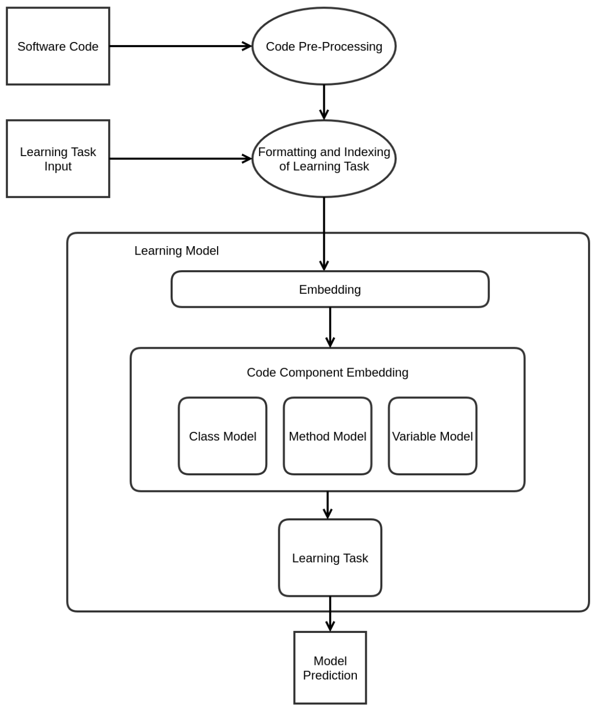


Figure 6.4: Learning Model

sion is treated as a sequence of code element tokens and sent through an RNN whose final output will be used as the embedding of the expression. This expression embedding is then merged with the modifiers (type, access, final, static, etc.) using an FFNN to produce a final vector that is used as the embedding of that variable.

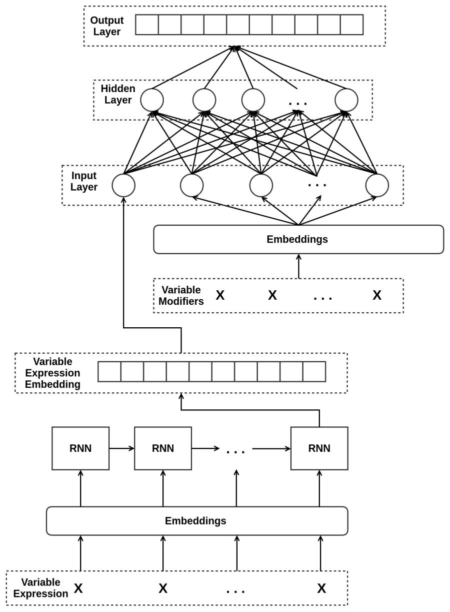


Figure 6.5: Variable Embedding Model

Method Model

Similar to the Variable Model, the Method Model (Figure 6.6 embeds a method name by collecting its method body and modifiers. The method body is treated as a sequence of code element tokens and sent through an RNN. The output of the RNN is merged with the modifiers (return type, access, final, static, etc.) using an FFNN to produce an embedding vector for that method.

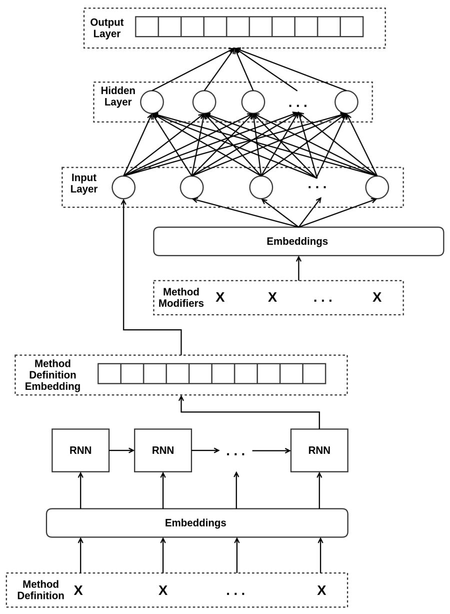


Figure 6.6: Method Embedding Model

Class Model

The Class Model, Figure 6.7, is a bit more sophisticated as it contains a variable number of members: inner classes, methods, and field variables. Further, what each type of member (inner class, method, and field variable) adds to the overall meaning of the classes differs. So each type of member and the modifiers are collected. Since there is variable numbers of members, for each type of member, the members are sent through a small FFNN. The output of all the members of a type are then summed together and then a final small FFNN combines the three separate types. This is very similar to an RNN, but without a transformation from hidden layer to hidden layer between each timestep, as the members of a class are not in sequence. This final vector is then merged with the class's modifiers using another FFNN and produces the final embedding for the class.

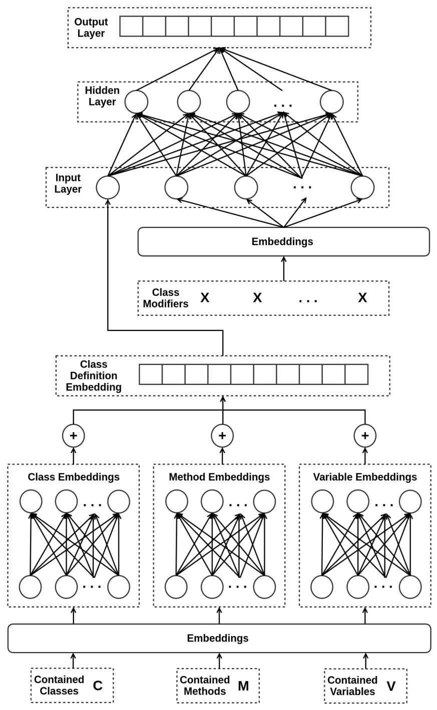


Figure 6.7: Class Embedding Model

Learning Vector Representations

The normal approach to learning vector representations is to perform a classification learning task, by either predicting the index number (or "class") of a word in the vocabulary (such as in N-Gram or Skip-Gram algorithms) or performing predicting if a pair of words is contextually related (such as in Word2Vec) by classifying the pair as "related" or "unrelated".

However, in the novel learning model approach just described this type of classification will be difficult. Therefore we will use a different pre-training learning task. First, initialize all word embeddings to random values. Then, for all code elements that have a declaration or definition, produce an embedding directly using the appropriate model. Then derive a loss based on the distance (e.g., Mean Squared Error) between the embedding produced by the model and the currently assigned embedding (that was initialized randomly) in the vocabulary. This loss will be used during propagation to modify and learn the model weights as well as the embeddings.

Figure 6.8 plots embeddings pre-trained from the novel learning model approach. As can be seen, we have achieved far superior clustering of code elements, implying that we have produced

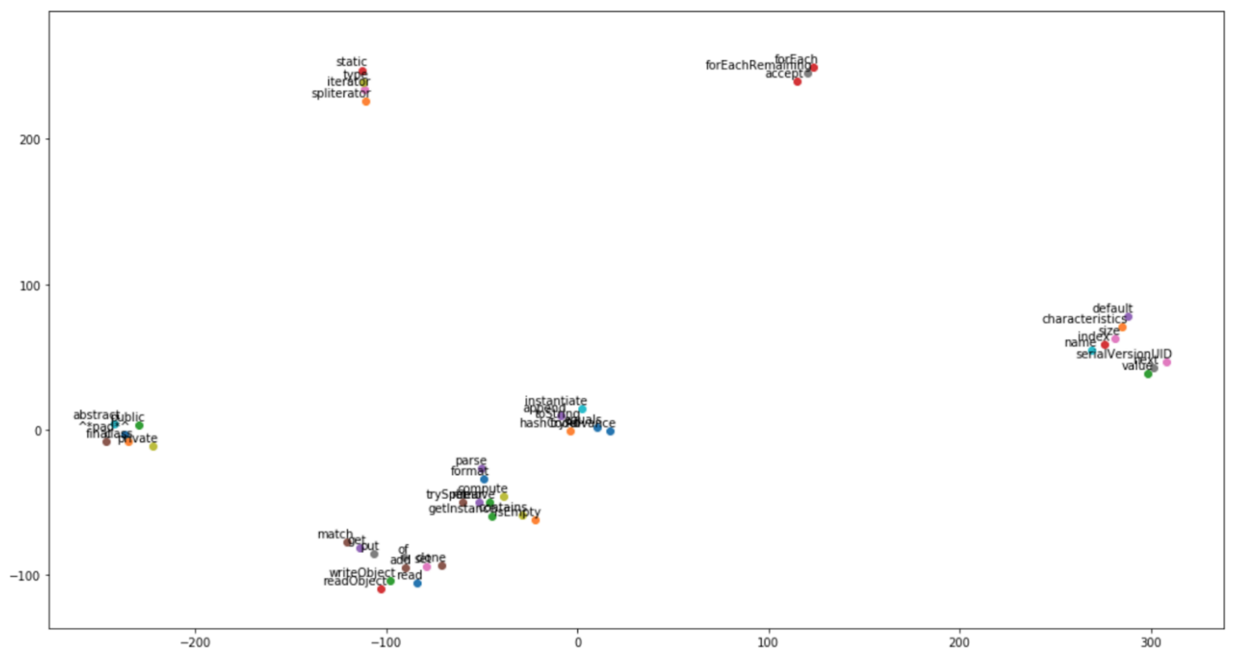


Figure 6.8: Vector Representations for the 45 Most JDK8 Code Elements

high quality vector representations.

CHAPTER 7: AUTOMATION OF SOFTWARE PRIVACY ANALYSIS

7.1 DICTA: Dynamic-Analysis-Induced Information-Type Classification of Call Traces in Android Apps

Mobile applications are widely used and often process highly sensitive data, such as a user's location, calendar data and other personal data. To assess how private data has been processed, many static and dynamic analyses tools are applied to detect information flows. To use those tools, we need to manually provide a list of sources of sensitive data and sinks where that data may leak, undesirably. To help us collect these lists, tools have been developed for automatically identifying the sensitive sources and sinks in the Android API. However, a user's sensitive data can be collected through various ways beyond the Android API, such as back-end servers or third-party services. We thus propose a deep-learning-based approach for identifying sources from the app code.

We trained the model using the stack trace collected via run-time testing on training apps and evaluated the trained model on the static call traces from new testing apps. The evaluation result shows that our approach identifies 81 sensitive call traces from testing apps and 22 of them can be validated via run-time testing.

As regulations [2, 18, 59] on user privacy increase in number and enforcement, app producers need to comply by accounting for how they process sensitive information and accurately describing their practices in privacy policies. There has been significant research toward this goal. Existing approaches [63, 66] mainly focus on platform information extracted from the mobile phone with Android system APIs. These approaches first identify a list of Android API methods which return sensitive information (e.g., *Android.telephony.TelephonyManager.getDeviceID()* returns the device ID of the phone), and then use static and dynamic analysis [7] to find out how the information flows in the app. More recent work [**?**, 56, 76] further accounts for user input data, thus tracking the execution of GUI API methods (e.g., *android.widget.EditText.getText()*) as potentially sensitive information sources. This includes classifying the data types by analyzing the GUI hierarchy [65] and labels associated with the method invocations.

However, these approaches still have two major limitations in locating all sensitive information processing in Android apps. First, Android API and user input do not cover all possible sources of sensitive information. An app may acquire data from back-end servers (e.g., user profiles which were collected using web portals or hand-written registration forms), and then share this data with other entities through the app. An app may also fetch data from third-party (e.g., Facebook profiles) services. Second, both dynamic analysis and static analysis are unsound in practice. Dynamic analysis only covers information flows observed at run time, and thus is limited by test coverage. While static analysis is theoretically sound, it is far from sound in practice due to dynamic code features (reflection, dynamically loaded code), external data flows (through database system, file systems and network), code heterogeneity (native code, multi-language code), and flaws in the static analysis design and implementation, such as modeling complex data structures. In our analysis, 20% of dynamically detected sensitive data are detected by static analysis, and only 21% of statically detected sensitive data are covered by dynamic analysis, thus illustrating the limitation of both approaches. Consequently, sensitive data flowing inside Android apps can be missed by existing approaches.

To better trace sensitive information in Android apps and supplement existing approaches, we propose a novel approach that uses deep learning to identify call traces that extract sensitive data from application data structures, and classify these call traces by the information types of the extracted data. Identified sensitive call traces can be further used as sources for static and dynamic taint analyses.

Most sensitive data is stored in memory as string variables. Although they are sometimes organized as fields into objects, there is often a public or private method defined in the object's class to fetch the value. Therefore, we can monitor all the return values of String type to identify sensitive data at run time. If the return value of a String-type method contains sensitive data of certain type T, we label the method as extracting sensitive data, and label its data type as T. Based on this intuition, we are able to construct a large, labeled training data set by automatically

```
xxxx@gmail.com
at com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:2)
at com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:3)
at com.tubitv.helpers.UserAuthHelper.getEmail(UserAuthHelper.java:1)
at com.tubitv.helpers.ZendeskHelper.setIdentity(ZendeskHelper.java:2)
at com.tubitv.helpers.LoginHandler.signInSuccess(LoginHandler.kt:10)
at com.tubitv.activities.SignInActivity.onSignInComplete(SignInActivity.java:6)
```

Figure 7.1: Example Stack Trace

exploring many Android apps and monitoring return values of String-type methods.

Data types of String-type methods can be determined only with their calling context. For example, given the stack trace in Figure 7.1, it is impossible to determine the data type of the *PreferenceHelper.getString(...)* method at the lowest level. This method is used across multiple contexts returning either sensitive or insensitive data. Together with the calling context (*UserAuth-Helpr.getEmail(...)* and *LoginHandler.signInSuccess(...)*), we can infer that the whole call trace fetches an email address. Based on this intuition, instead of labeling a single method, we label the information type of a call trace within a specified length. In our preliminary study, we found that, within six levels of calling contexts, more than 90% of observed call traces have determined a data type (i.e., were observed to fetch only one type of data at run time). So we consider call traces within length six in our implementation and evaluation.

In order to classify the stack traces by information type, we constructed a data set of 214,106 stack traces (note that we consider only stack traces whose method on top returns a String) collected at run time during testing of 68 apps (top Android apps). Each stack trace is labeled with the information type of its run time return value. For each information type, we randomly select apps to the training set until the apps in the training set contain 80% of the stack traces, and then we use the remaining apps as testing apps. After that, we use the stack traces of apps in the training set to train a model. The learned model was utilized to classify call traces from the static call graphs of the testing apps to evaluate the effectiveness of the trained model. Similar to the labeling of training data, we thoroughly tested the testing apps and label a call trace based on the information type of its return value.

Dataset

Due to the testing coverage limitation about dynamic analysis, the goal of this approach is to identify sensitive call traces which can not be observed during run time. DICTA utilizes the ground truth collected by dynamic analysis to train a model. For a given unclassified call trace extracted from a call graph, DICTA will decide whether it is a sensitive trace or not. In this section, we explain the details of DICTA. Similar to standard natural language processing techniques, there are three main components to our model design. The first component describes data collection and selection for training, validation, and testing sets; the second component explains the conversion of call traces into a format that can be processed and learned from by a machine learning model; and the final component details the machine learning task and model for information-type classification.

In order to collect the stack traces, we perform value-based dynamic taint analysis, which includes the following steps:

- Building a user profile. Value-based analysis requires uncommon tainted values as sources. Specifically, we use the values in the user profile of an Android device as the tainted values. The profile includes different platform IDs (e.g., device ID, serial number, Android ID, advertising ID), and a synchronized Google account (e.g.,user name, full name, user email). The list of values in the user profile are presented in Table 7.1.

- Instrumenting on apps. We use Apktool [80] to decompile an APK file into smali code. In order to catch the return value of String-type methods during runtime, we instrumented all String-type methods in the smali code by adding a call to the Android logger to report the invocation at the beginning of the method implementation. This allows us to use the Android system log to analyze method return values set at runtime.

Table 7.1: User Profile

Info Type	Value
AndroidID	"a54eccb914c21863"
Email	"********@gmail.com"
AdvertiserID	"fc1303d8-7fbb-44d8-8a68-a79ffac06fea"
User Name	"******"
IMEI	"355458061189396"
Serial	"ZX1G22KHQK"
Password	"******"

- Run-time testing on app. After inserting the code, we rebuilt the smali code back into APK format for testing. We used the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device and ran the apps while executing Monkey [21] to perform the testing. For each app, we automatically installed, executed, tested, uninstalled and saved the system log into the local file system for later inspection. During testing, some apps requiring registration and login to show the app's start page, so we manually created accounts for those apps using the predefined user profile to complete the login process. DICTA is then able to identify sensitive traces by searching for the values in the profile.

Utilizing the Androguard framework [20], we disassemble apps and extracted their function call graphs. Each extracted call graph is a directed graph, which contains a list of nodes that represent each of app's functions and edges indicating each invocation from caller to callee. In order to collect the call traces, we first collect the String-type methods. Then, for each collected function, we traverse the call graph starting with the corresponding function node and traverse backward 6 steps towards the root (i.e., the program entry point) for each possible branch. By doing so, we are able to extract a list of call traces for each String-type method.

7.1.1 Learning Model for Stack Trace Prediction

Learning Models

Call traces are a list of fully qualified method names denoting the order the methods were called and executed in. These method names are usually composed of words from natural language (e.g., *android.location.Location.getLatitude()* is the method *getLatitude()* in the class Location in the package android.location). Since machine learning models are not usually able to process words directly, we must first convert those fully qualified method names to some numeric form by extracting and constructing representative features for them. For this model we have chosen the bag of words (BOW) approach, which identifies the individual words (or segments) that the fully qualified method name is composed of as features. These words are used to construct a vocabulary that can be used to create a vector representing which words compose the fully qualified method name.

In order to implement BOW, we created a vocabulary list of all known words used in the fully qualified method names of the call traces (e.g., *android.location.Location.getLatitude()*). We began by removing parameters from the end of the method name (e.g., *android.location.Location.getLatitude*) and then splitting each fully qualified method name on the dot operator (.) and separating the package name, class name, and method name (e.g., *android.location.Location.getLatitude()* would become package name: *[android,location]*, class name: $[Location]$, and method name: *[getLatitude]*). Then, each part of the name was split on the punctuation allowed in android names (i.e., _ and $) and was further split into smaller words using a regular expression for the camel case naming convention very common in android code (e.g., package name: *[android,location]*, class name: *[Location]*, and method name: *[getLatitude]* would become package name: *[android,location]*, class name: *[Location]*, and method name: *[get,Latitude]*. These words were then inserted into the vocabulary, each word converted to all lowercase and tagged as coming from the package, class, or method part of the fully qualified name.

The BOW vector representing the fully qualified method name is constructed by instantiating a vector that is the length of the vocabulary. Each index in the vector represents the word at the same index location in the vocabulary list. Each element in the vector is first initialized to zero, and then for each word that composes the fully qualified method name (obtained by the exact same process as when building the vocabulary) the vector element is incremented at the index equal to the index of the word in the vocabulary list. For example, given the vocab-

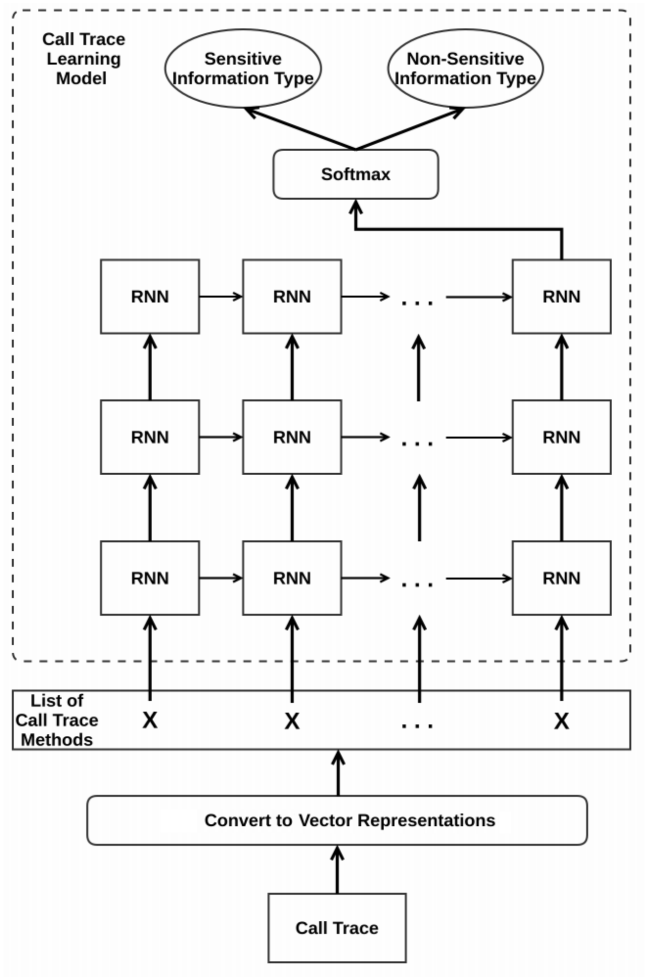

Call Trace Learning Model
Sensitive Information Type
Non-Sensitive Information Type
Softmax
RNN
. . .
List of Call Trace Methods
X
Convert to Vector Representations
Call Trace

Figure 7.2: Stack Trace Learning Model

ulary list: *[java(package), android(package), location(class), com(package), location(package), get(method), set(method), location(method), latitude(method)*, the BOW vector representation for the method *android.location.Location.getLatitude()* would be: $[0, 1, 1, 0, 1, 1, 0, 0, 1]$.

The learning task is, given a call trace, classify what type of sensitive (or some non-sensitive) information type is returned by the call trace. The normal approach would be to construct a multi-class learning model that would predict between the different sensitive information type classes plus one class to represent all non-sensitive information types (or all information types not represented by the other defined classes). However, there are two main issues with this approach. The first is that our data set for training the learning model is unbalanced, with very few call traces returning one of the sensitive information type classes and very many call traces returning some non-sensitive information type, leaving limited data for training. The second issue is the multi-label problem as a single call trace can return multiple different classes of sensitive information types. In order to help mitigate these issues, we have chosen to break down the learning task into separate binary learning problems. That is, we will create separate learning models, one for each individual sensitive information type, and perform a binary classification task between the sensitive information type class and non-sensitive information type class. Doing so eliminates the multilabel classification problem and will be able to more effectively learn over the limited data set.

We have constructed a recurrent neural network (RNN) model for our learning task. The RNN model was most appropriate as it excels at processing sequences of data, and a call trace is a sequence of method calls.

As show in Figure 7.2, the input to our call trace learning model is a sequence of vector representations (X) derived from a call trace. The method sequence is in the order in which the methods were called during program execution. This sequence of vector representations is given to the learning model as input. The model is composed of a three-layer RNN, with a final softmax prediction for the two classes "Sensitive Information Type" or "Non-Sensitive Information Type". Cross-entropy is used for our loss function for learning.

After training, the model was run on the call traces in the testing apps. As shown in Table 7.2, while the model achieved over 90% in correctly predicting non-sensitive traces (recall), its performance only correctly predicted 21% sensitive traces (precision). We analyzed the false positives of the model and found that many of the traces were heavily obfuscated and contained may methods with the similar names as those in non-sensitive traces (e.g., *toString*, *toJson*, etc.). This will make it very difficult for a model to learn based solely on the names of methods in the stack traces.

FastText [12, 41] is a popular open-source text representation and text classifier library. It has implementations based on BOW or Skip-gram models. We used the Skip-gram implementation to further compare other types of models. The pre-processing of method names in the stack traces are performed in the same ways as the BOW model. FastText is able to then take the set of word tokens and produce a single embedding (or vector representation) for the entire method. These embeddings are then input to the model as shown in Figure 7.2.

In order to learn higher quality vector representations for each method in the stack trace, we apply our novel learning approach detailed in Chapter 6. The pre-processing and preparation of code for embedding is performed as described in Section 6.2.1. The RNN model for learning and predicting stack traces is the same as previously described in this chapter, however the model has been extended to include an additional layer of each method in the stack trace being embedded, on-the-fly, based on the method component embedding described in Section 6.2.2.

Results

Table 7.2: Results of applying the different models to the testing set

	Recall	**Precision**	**F1 Score**
BOW Model	0.9	0.21	0.34
FastText Model	**0.99**	0.34	0.51
Novel Learning Approach	0.76	**0.55**	**0.64**

As shown in Table 7.2, while the FastText was able to better predict non-sensitive traces, the novel learning approach was able to predict sensitive traces at over 20% of FastText's ability and an improved F1 score of over 10%.

CHAPTER 8: EVALUATION

For our evaluation, we will reference the thesis statement:

Deep learning techniques can be effectively applied to static code analysis by introducing a new semantic learning approach that will be able to overcome the limitations of traditional static analysis and current deep learning techniques by utilizing the equivalence relation between code elements and their declarations and definitions.

To determine the validity of this statement, we will revisit the previous results obtained throughout this work.

8.1 Vector Space Analysis

In Figures 8.1 and 8.2, we see the same code element vector representations plotted in their vector space from Chapter 6. Figure 8.1 used a novel approach to identifying context based on which code elements actually affected its behavior instead of the traditional method of identifying only immediately surrounding code elements. Figure 8.2 used the novel learning approach to produce its vector representations. In both figures, the best clusters have been circled. It would appear obvious that the vector representations from the novel learning approach have much improved clusters over that of the model based on the traditional learning approach.

This improved clustering implies that the model is able to produce high-quality vector representations for code elements. Such vector representations are often necessary in order to gain any degree of success in NLP techniques. We also have some confidence that the data sparsity problem has been effectively mitigated, as each vector representation was produced by only using the single declaration or definition of that code element. It may be argued that many code elements learned this way were seen more than once as part of the declaration or definition of other code elements. This is true, however such quality code element vector representations were achieved using the same number of occurrences as the other techniques that did not perform as well. This indicates that this approach is able to learn more effectively with less code element occurrences. It is also

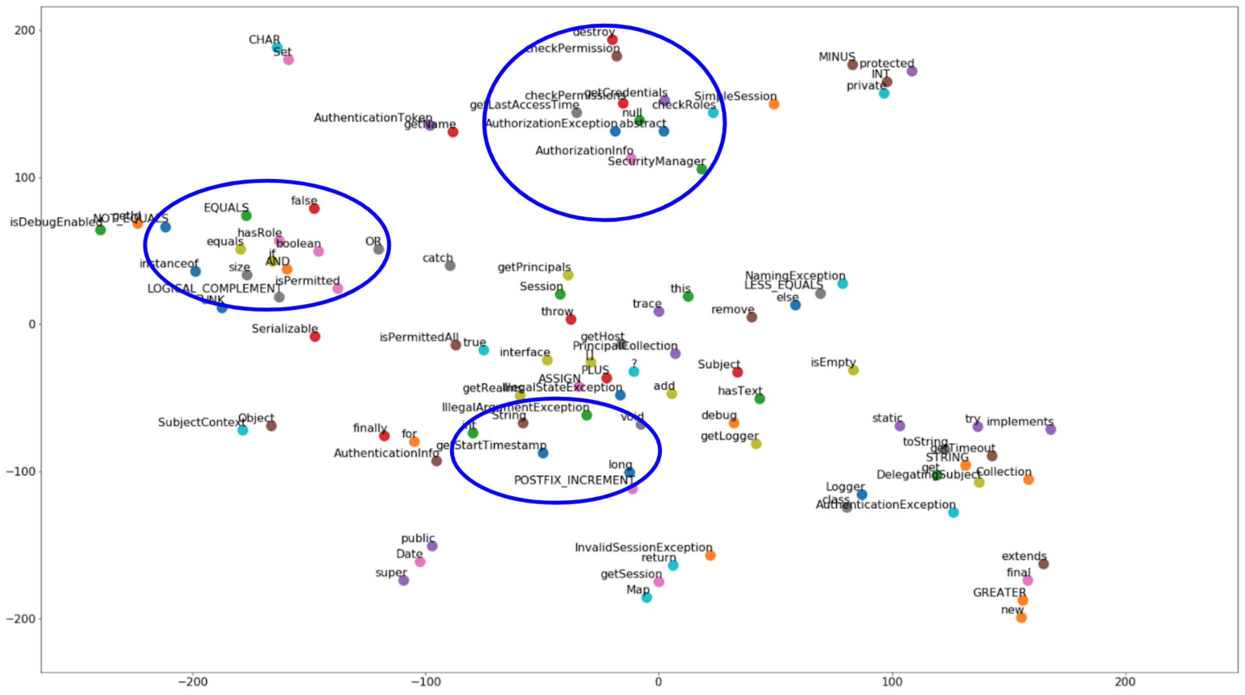


Figure 8.1: Clustered Vector Representations for the 99 Most Common Apache Shiro Code Elements

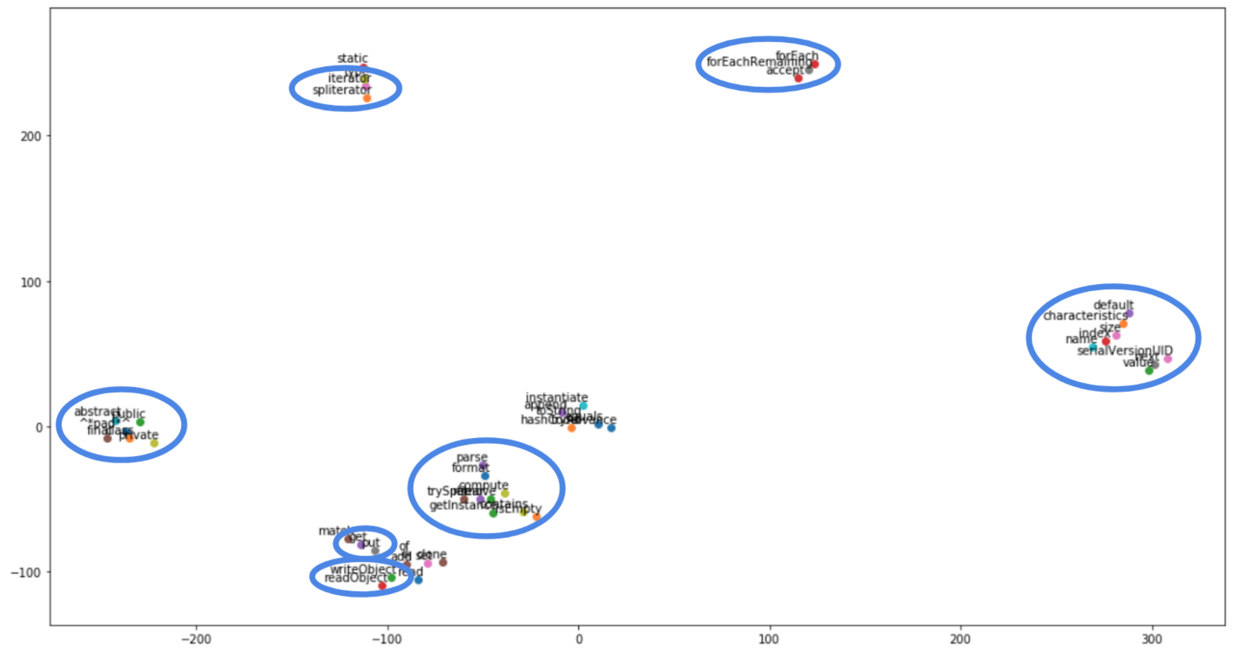


Figure 8.2: Clustered Vector Representations for the 45 Most JDK8 Code Elements

worth noting that the data sparsity problem is not fully mitigated. There are code elements that have no declaration or definition (e.g., keywords, syntactic symbols, operators, etc.). These code elements are still subject to the data sparsity problem and can only be learned implicitly during the learning backpropagation of other code elements. However, in Figure 8.2, such code elements have been plotted (e.g., public, private, final, etc.) and still show effective clustering.

As we are able to achieve these quality vector representations, the effectiveness of deep learning should be significantly increased.

8.2 Privacy Compliance Results

Though Chapter 5 does not deal with software code directly and instead only with privacy policies, it is still worth noting that the automation of the ontology maintenance was generally successful. While the Experiment 1 results from Table 5.1 did not perform as well as hoped, Experiment 2 in Table 5.2 performed at the state-of-the-practice of natural language processing deep learning tasks. Further, Experiment 2 encapsulates Experiment 1. Likely, the greater success of Experiment 2 was due to the increased number of hypernymy examples to allow for more learning opportunities. This work shows that there is potential in applying deep learning to NLP static analysis problems for automation and generalization of tasks. Further, though this was specific to normal natural language, and not software code, as discussed earlier software code can be treated as a natural language. While software code has some unique problems, the more success in deep learning on normal natural language problems means greater potential for success in deep learning on code element natural language problems.

In Chapter 7, we compared the results of a BOW model and the novel learning approach. BOW achieved 90% recall (i.e., correct prediction of non-sensitive traces) and 21% precision (i.e., correct prediciton of sensitive traces), and the novel learning approach achieved 75% recall and 55% precision on the testing dataset. While the novel learning model did not perform as well on the prediction of non-sensitive traces, it more than doubled the prediction results of sensitive traces. Though the results could be improved, it is worth noting the testing dataset was very unbalanced in favor of non-sensitive traces (a few thousand non-sensitive traces to a few dozen sensitive ones). Further, there are additional techniques that can be applied to novel learning model to help improve its results. So the results do show promise and potential.

8.3 Discussion

Though no complete model for privacy compliance verification was achieved, the application of deep learning to improve some sub-problems of static analysis was successful. Though there is still much work to be done to replace or drastically assist static analysis as the state-of-the-art for privacy compliance, there is promise in many of the projects and techniques presented.

Based on the results of the projects contained in this work, it can be confidently declared that there is great potential in the application of deep learning to static code analysis. We have shown that the novel learning approach was effective in helping to mitigate some of the limitations of deep learning natural language techniques and some of the unique obstacles of deep learning on software code, and that the automation and generalization of static analysis can be assisted by the application of deep learning techniques.

CHAPTER 9: CONCLUSION

In this work we have discussed the importance of privacy compliance in software analysis. We have seen that static analysis, the state-of-the-art, is capable of performing compliance analysis between privacy policies and software code. However, the analysis suffers from many limitations that affect the ability to automate and generalize the analytical models. We showed that it is possible to overcome such obstacles through the application of deep learning models, especially through the use of natural language processing techniques.

We were able to successfully apply deep learning to privacy policy through the use of current deep learning NLP techniques. However, it was shown that software code had additional problems that such techniques could not easily overcome, such as: more complex syntactic structures, an increase in the number of new code elements encountered during training and testing, and a far more severe data sparsity problem.

By taking advantage of the equivalence relationship between a code element and its declaration or definition, we were able to construct a novel learning approach to mitigate the obstacles of deep learning on software code. By utilizing the novel approach, we were able to produce higher quality vector representations for code elements and an increase in precision and recall for some deep learning prediction tasks.

9.1 Future Work

9.1.1 Defect Detection and Measuring Software Reliability

We plan to expand on the application of this work by applying our models and techniques to defect detection. Based on previous work [26] that describes a new measurement model based on software code defect detection. Since it is based on static analysis, it suffers from many of the same limitations as privacy analysis. We hope to be able to apply our novel learning approach to defect detection to help automate and generalize our reliability measurement model.

The security and vulnerabilities of software systems are essential in making decisions for real-world problems. Many process-based approaches and models have been developed to mitigate and quantify software vulnerabilities. However, these approaches, while useful, have many limitations. First, the existing models require data from software processes to estimate model parameters, but fine-grained monitoring of software processes is not always possible. Second, these models cannot always handle new or upgraded software components. Third, these models often cannot differentiate between different types of defects. To address these issues, the proposed project will develop a pattern-based vulnerability measurement model, which checks software artifacts for the existence of negative patterns to estimate the risk of software failures and data security, their impact, and determine overall software vulnerability.

The model will quantitatively estimate the vulnerability of software based on negative pattern instances in software artifacts, as well as the importance and the activation probability of the patterns. Recent studies show that software reuse and code clones are prevalent throughout software systems. In addition, software projects are continually becoming more based on existing software frameworks, which have a limited number of usage patterns. Therefore, it is reasonable to assume that most of the design fragments and code portions of a new software product follow existing patterns. Furthermore, the vulnerability of a software application can be predicted by combining effects of all instances of negative bug patterns. The project will yield a **learning engine** to mine online bug repositories and project hosting websites; a **pattern checker** to detect the existence and invocation of patterns; and a **vulnerability model** for the estimation of the vulnerability of a software project based on the detected patterns and their invocation probabilities.

The project will achieve the following major objectives: 1) Quantitatively estimate the vulnerability of a software project based on code patterns; 2) Support to separate estimation of different aspects of software vulnerability, which enables fine-grained prediction to the effect of software failures; 3) Confirm the existence of negative patterns (and thereby confirm the existence and location of access control bugs) using test-coverage-based approach; 4) Evaluate the model on real-world software projects to conduct a feasibility study of the model's ability to perform software vulnerability estimation.

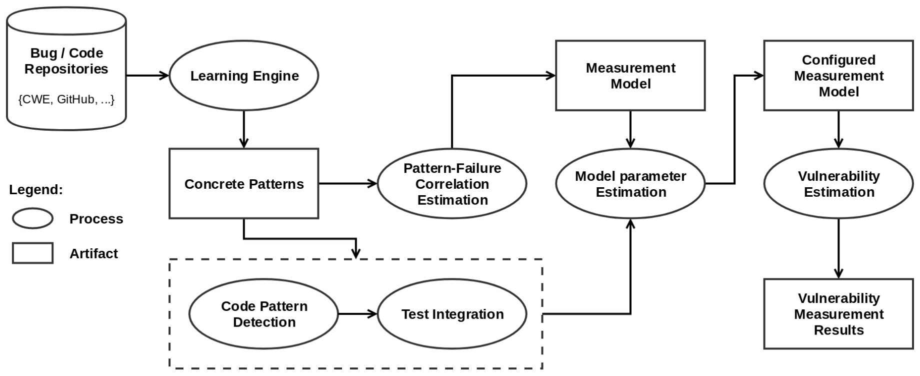


Figure 9.1: Bug Pattern Framework

Learning Engine

The learning engine acts as a repository of known patterns. These patterns will be used to identify bugs in code during pattern detection. To create this initial repository, we first surveyed popular online bug repositories, such as Github, Bugzilla, Common Weakness Enumeration (CWE)[1], Jira, etc. We decided to initially support CWE and Github, as they often have code examples and code fixes linked with their bug reports. Further, they both offer very large, robust data sets of access control patterns and bugs.

CWE is a database that catalogs and categorizes known bugs. Our method crawls bug reports from CWE and categorizes them according to the common keywords in their descriptions and CWE classification categories. We are thus able to extract bug descriptions, code examples, and code solutions.

The goal of bug collection for Github is to identify the most common bugs on Github. We obtain thousands of issues from Github projects by making requests to Github's REST API v3. For each of these issues we are able to extract bug descriptions, solutions, and sometimes sections of code from before and after an issue was resolved. In our initial implementation we searched the top 1,000 Java-based repositories, and used the most recent 100 closed issues with the label "bug" from each. After we collected the data from CWE and Github, we applied the clone detection tool CCFinder [42] to detect clones from the set of code commits and identify common buggy code and

[1]CWE website URL: https://cwe.mitre.org/

their fixes. Then, for the most popular common code bugs we manually extracted concrete code patterns from them, which were stored to be used in pattern detection and code analysis.

Pattern Detection

There is limited literature available on bug pattern detection. Work by Joseph Near and Daniel Jackson [57] describes the tool "SPACE" which is able to detect violations of access control vulnerability patterns. However, this tool requires that a user completely define and map software components and design to the role-based access control model. Not all software systems specifically use role-based access control, though. It can be difficult, or not possible, for systems that use other access control designs to define that system in the role-based model. Further, this approach is based on seven access control patterns, not code-level bug patterns on access control. Other work by Guangtai Liang et al. [43] introduces the tool "PatBugs" which detects temporal bugs in cross-platform mobile applications, focusing on API usage. The scope of this tool is too narrow for our purposes, it's focus being solely on temporal bug patterns in mobile applications.

After identifying bug patterns, they further needed to be linked to abstract quality aspects to calculate a vulnerability, or risk, measure. Specifically, we consider four major aspects of vulnerability including: (1) **Control Integrity** which measures how likely the software may incorrectly interact with its users; (2) **Data Integrity** which measures how likely the software may provide incorrect output; (3) **Data Confidentiality** which measures how likely the software may release data to entities not authorized to receive it; and (4) **Data Availability** which measures how likely the software may not be able to provide data that should be in storage. Through linking bug code patterns with high-level quality aspects, we are able to estimate the vulnerability, or risk, on different aspects based on detected patterns related to them.

Finally, the detection of a bug pattern is not enough to determine that a bug actually exists, only that a bug possibly exists. We integrate testing techniques in order to test if the section of code identified by the pattern actually produces an error, showing it truly is a bug.

FindBugs

To conduct the pattern detection in our framework, we utilized SpotBugs[2], a fork of the static analysis tool FindBugs [8]. The tool analyzes Java bytecode and detects the existence of bug patterns.

To study the effectiveness of SpotBugs on detecting bug code patterns, we downloaded the top 1,000 Java GitHub repositories that are under 50MB for a total of 826 repositories. From these repositories, we were able to compile 763 repositories using a program for automatic compilation of various types of Java projects. For each compiled repository, we ran SpotBugs to generate a list of bug types, which detected 217 bug pattern instances.

Measurement Model

The model used to calculate a vulnerability score must consider as many different types of bugs (related to access control) as possible, and must cover the abstract quality aspects mentioned previously. We began by considering existing measurement models (e.g., Common Vulnerability Scoring System (CVSS) [49], Common Weakness Scoring System (CWSS) [3], etc.), however these models only calculate a scoring for a single bug and not for an entire system. Further, the amount of possible automation in these models is also very limited and requires manual parameter assignments. Therefore, we created our own measurement model.

The basic idea of our measurement model is to estimate the vulnerability of a software based on the detected instances of code patterns in its code base. For a bug pattern instance, we determine the impact it will have on the software in relation to the identified abstract quality aspects. Further, we determine how likely the instance will be triggered at runtime, which was found from the testing performed previously. The more instances of bug patterns detected and the more likely those instances will be triggered, the higher (or worse) the vulnerability score should be.

At the most abstract level, for multiple detected instances of bug patterns, we use the following

[2]SpotBugs URL: https://spotbugs.github.io/
[3]CWSS URL: https://cwe.mitre.org/cwss/cwss_v1.0.1.html

formula to generate a vulnerability value, normalized to the range [0, 1]. Here *Detected* is the set of bug instances detected in the code using patterns and *Risk(b)* denotes the risk value of a given bug *b*.

$$Vulnerability = 1 - \frac{R}{R + \sum_{Detected} Risk(b)} \tag{9.1}$$

R is a constant, which is the average risk sum per software project, which can be estimated using a large number of training software projects. With this formula if there are no bugs in a software project, the vulnerability score will be 0. If the risk sum of all bugs in a project is *R*, the vulnerability score will be 0.5. So vulnerability scores above 0.5 indicate above average vulnerability, and lower than 0.5 indicate below average vulnerability. When the risk sum goes very high, the vulnerability value will be close to 1.

The current, top-level formula for the risk of a bug is:

$$Risk = Impact * Susceptibility \tag{9.2}$$

Impact represents how the behavior and data of the software are affected by the bugs present in it. *Susceptibility* defines how easy or often those bugs are executed.

We divide *Impact* into four different sub-aspects: Control Integrity, Data Integrity, Data Confidentiality, and Data Availability, which cover our abstract aspects. They are modeled by the equation:

$$\begin{split} Impact = A * Integrity_{Control} + B * Integrity_{Data} \\ + C * Confidentiality_{Data} + D * Availability_{Data} \end{split} \tag{9.3}$$

A, *B*, *C*, and *D*, model the relative importance (or weight) of $Integrity_{Control}$, $Integrity_{Data}$, $Confidentiality_{Data}$, and $Availability_{Data}$, respectfully. The weights of different aspects may be changed according to the actual usage scenarios. It should be noted that, although the formula currently models only negative patterns, positive patterns and mitigations can also be considered in the same way by changing the *Impact* value in a negative way.

Susceptibility indicates how likely the bug may be triggered during runtime. That is, when a software is executed, a bug that is triggered very often is more of a risk than a bug that is triggered rarely. *Susceptibility* can be estimated using the results from the integrated testing. The more tests that trigger the bug, the higher the *Susceptibility* should be.

For our framework, we have created a website[4] where a Github repository URL can be entered. If the automatic build script is able to successfully build the repository, it will produce measurement results and bug detection outputs based on our framework. For the final vulnerability calculation, *R* has been determined based on the average bug occurrences across 717 top Github Java projects.

Toward the Application of Deep Learning NLP Techniques

Like most sophisticated vulnerability and reliability measurement frameworks we found, many processes in our framework must be performed manually. Since the magnitude of these manual processes makes such frameworks infeasible for real-world use, we hope to utilize our novel learning approach to mitigate some or all of the manual processes to make our framework practical. So, our learning task for this project will be defined as, given a method's declaration and definition, predict if the method contains a defect (or bug).

To perform such a task, we will need to modify the current learning model used for stack trace prediction in Chapter 7. The current component learning models (i.e., class model, method model, and variable model) will be directly applicable to producing vector representations for code elements in the input method. However, the stack trace prediction component will need to be modified to learn over an entire method body instead of just a stack trace. Such a change should be trivial to implement.

[4]http://galadriel.cs.utsa.edu:25666/

9.1.2 The Novel Learning Approach for Natural Language

We have shown how the novel learning approach can be effectively applied to code. This approach can also be utilized for normal natural language. Each word in natural language has an associated definition, and can be utilized by the novel learning approach in the same way as a declaration or definition of a code element. Other word characteristics (e.g., part of speech) would take the place of code element modifiers. Just as in code, this approach should help mitigate limitations of natural language processing on natural language, such as: the data sparsity problem, encountering new words during testing, etc.

BIBLIOGRAPHY

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

[2] Accountability Act. Health insurance portability and accountability act of 1996. *Public law*, 104:191, 1996.

[3] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

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

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

[6] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013*, 114:108, 2013.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[8] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[9] Adam Barth, Anupam Datta, John C Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *2006 IEEE symposium on security and privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[10] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[11] Jaspreet Bhatia and Travis D Breaux. Towards an information type lexicon for privacy policies. In *2015 IEEE eighth international workshop on requirements engineering and law (RELAW)*, pages 19–24. IEEE, 2015.

[12] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[13] Travis D Breaux, Annie I Antón, and Eugene H Spafford. A distributed requirements management framework for legal compliance and accountability. *computers & security*, 28(1-2):8–17, 2009.

[14] Travis D Breaux, Daniel Smullen, and Hanan Hibshi. Detecting repurposing and over-collection in multi-party privacy requirements specifications. In *2015 IEEE 23rd international requirements engineering conference (RE)*, pages 166–175. IEEE, 2015.

[15] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[16] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H Winsborough. Privacy promises that can be kept: a policy analysis method with application to the hipaa privacy rule. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 3–14, 2013.

[17] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[18] Federal Trade Commission et al. Children's online privacy protection act of 1998 (coppa), 1998.

[19] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2018.

[20] Anthony Desnos et al. Androguard, 2011.

[21] Android Developers. Ui/application exerciser monkey, 2012.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[24] 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. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[25] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G Shin, and Karl Aberer. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 531–548, 2018.

[26] John Heaps, Rocky Slavin, and Xiaoyin Wang. Toward a code pattern based vulnerability measurement model. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 209–211, 2018.

[27] John Heaps, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. Toward detection of access control models from source code via word embedding. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, pages 103–112, 2019.

[28] John Heaps, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. Toward detection of access control models from source code via word embedding. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, pages 103–112. ACM, 2019.

[29] John Heaps, Xueling Zhang, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. Toward a reliability measurement framework automated using deep learning. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, pages 1–2, 2019.

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

[31] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

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

[33] Mitra Bokaei Hosseini, Travis D Breaux, and Jianwei Niu. Inferring ontology fragments from semantic role typing of lexical variants. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 39–56. Springer, 2018.

[34] Mitra Bokaei Hosseini, Sudarshan Wadkar, Travis D Breaux, and Jianwei Niu. Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series*, 2016.

[35] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[36] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 977–992, 2015.

[37] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, pages 1606–1612, 2016.

[38] I Jet Brains. Intellij idea. *On-line at www. intellij. com*, 2011.

[39] Claiborne Johnson, Thomas MacGahan, John Heaps, Kevin Baldor, Jeffery von Ronne, and Jianwei Niu. Verifiable assume-guarantee privacy specifications for actor component architectures. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 167–178, 2017.

[40] Bengt Jonsson and Tsay Yih-Kuen. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1-2):47–72, 1996.

[41] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[42] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[43] Guangtai Liang, Jian Wang, Shaochun Li, and Rong Chang. Patbugs: A pattern-based bug detector for cross-platform mobile applications. In *2014 IEEE International Conference on Mobile Services*, pages 84–91. IEEE, 2014.

[44] Sarah Lichtenstein and Paul Slovic. *The construction of preference*. Cambridge University Press, 2006.

[45] Thomas MacGahan. *Towards verifiable privacy policy compliance of an actor-based electronic medical record system: An extension to the HAPL language focused on exposing a user interface*. The University of Texas at San Antonio, 2016.

[46] Thomas MacGahan, Claiborne Johnson, Armando Rodriguez, Jeffery von Ronne, and Jianwei Niu. Provable enforcement of hipaa-compliant release of medical records using the history aware programming language. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 191–198, 2017.

[47] Steve McConnell. *Code complete*. Pearson Education, 2004.

[48] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[49] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6), 2006.

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

[51] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, L Sutskever, and G Zweig. word2vec. *URL https://code. google. com/p/word2vec*, 22, 2013.

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

[53] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19(88):2, 1969.

[54] MITRE. Common weakness enumeration: A community-developed dictionary of software weakness types. *MITRE Corp*, 2019.

[55] Bennet B Murdock Jr. The serial position effect of free recall. *Journal of experimental psychology*, 64(5):482, 1962.

[56] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 993–1008, 2015.

[57] Joseph P Near and Daniel Jackson. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering*, pages 947–958, 2016.

[58] NIST. National vulnerability database. *National Institute of Standards and Technology, U.S. Department of Commerce*, 2019.

[59] European Parliament and The Council of the European Union. General data protection regulation (gdpr), 2016.

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

[61] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[62] Gabriele Petronella. Analyzing privacy of android applications. 2014.

[63] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

[64] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[65] A Rountev, D Yan, S Yang, H Wu, Y Wang, and H Zhang. Gator: Program analysis toolkit for android, 2017.

[66] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 25–36, 2016.

[67] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub, oct. de*, 2017.

[68] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. Ernie: Enhanced representation through knowledge integration. *arXiv preprint arXiv:1904.09223*, 2019.

[69] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Hao Tian, Hua Wu, and Haifeng Wang. Ernie 2.0: A continual pre-training framework for language understanding. *arXiv preprint arXiv:1907.12412*, 2019.

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

[71] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.

[72] Duyu Tang, Bing Qin, and Ting Liu. Learning semantic representations of users and products for document level sentiment classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1014–1023, 2015.

[73] Tricentis. The 5th edition of the software fail watch. *Statistical Report*, 2018.

[74] Jeffery von Ronne. Leveraging actors for privacy compliance. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 133–136. 2012.

[75] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[76] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*, pages 37–47, 2018.

[77] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

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

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

[80] R Winsniewski. Android–apktool: A tool for reverse engineering android apk files. *Retrieved February*, 10:2020, 2012.

[81] Yichen Xie and Alex Aiken. Saturn: A sat-based tool for bug detection. In *International Conference on Computer Aided Verification*, pages 139–143. Springer, 2005.

[82] Wayne Xiong, Lingfeng Wu, Fil Alleva, Jasha Droppo, Xuedong Huang, and Andreas Stolcke. The microsoft 2017 conversational speech recognition system. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5934–5938. IEEE, 2018.

[83] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *2016 AAAI Fall Symposium Series*, 2016.

VITA

John Heaps was born and raised in Houston and Austin, Texas. He moved to San Antonio, Texas where he attended the San Antonio Community College and earned an Associate of Science in Mathematics before attending the University of Texas at San Antonio. There, he earned a Bachelors, Masters, and PhD in Computer Science. During his time at the University of Texas at San Antonio he served as a Tutor, Grader, Teaching Assistant, Research Assistant, and Lecturer. He also obtained an Internship at the Los Alamos National Laboratories. He has served on many projects dealing with software reliability and deep learning, and hopes to continue the vast and fascinating research areas relevant to Artificial Intelligence and Software Engineering.