Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Mathias Knutsen
Eivind Hestnes Lervik

Detection of Vulnerabilities in Source Code Using Machine Learning and Natural Language Processing

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2022

NTNU
Norwegian University of
Science and Technology

Mathias Knutsen
Eivind Hestnes Lervik

Detection of Vulnerabilities in Source Code Using Machine Learning and Natural Language Processing

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Vulnerability detection is not a new topic, but in recent years it has only become more important. As security requirements for software solutions become increasingly stricter, and the cost of development and testing only rises, there is a need to catch the vulnerabilities before production. The countermeasures in place today include heavy static and dynamic analysis, as well as time-consuming testing and fuzzing. Modern machine learning techniques have been applied on top of static analyzers, and as standalone solutions in order to contribute and take some weight off the shoulders of developers.

In this thesis, we review the literature on the topic of vulnerability detection using machine learning. Based on the literature, we propose several machine learning approaches to the problem. Our models train only on function snippets, without any outside context about the code to simplify the problem and improve processing speeds. Our best model successfully detects 70% of all vulnerabilities in a test set extracted from real-world source code, while producing fewer than one false positive for each real vulnerable function found.

Sammendrag

Påvisning av sårbarheter er ikke er nytt tema, men de siste årene har det bare blitt viktigere. Ettersom sikkerhetskrav for programvareløsninger stadig blir strengere, og kostnadene for utvikling og testing øker, er det nødvendig å finne sårbarhetene før produksjon. Tiltakene som er på plass i dag inkluderer tunge statiske og dynamiske analyseverktøy, samt tidkrevende testing og fuzzing. Moderne maskinlæringsteknikker har blitt brukt på toppen av statiske analyseverktøy, og som frittstående løsninger for å bidra til å lette på arbeidsmengden til utviklere.

I denne oppgaven gjennomgår vi litteraturen på temaet om påvisning av sårbarheter ved bruk av maskinlæring. Basert på litteraturen foreslår vi flere måter å løse problemet med maskinlæring. Våre modeller trener bare på kodesnutter av funksjoner uten noe kontekst om den omkringliggende koden for å forenkle problemet og minske behandlingstid. Vår beste modell oppdager 70% av alle sårbarheter i et testsett hentet fra kildekode i den virkelige verden, samtidig som den produserer færre enn én falsk positiv for hver ekte sårbarhet funnet.

Preface

This master's thesis is the very last part of our master's degree. Preceding this thesis, we completed a specialization project that concluded most of the literature search and review in this thesis. Software security has always been an interesting topic during our studies, and with some inspiration from our supervisor, this project was chosen. We wrote our bachelor's thesis on malware detection in executables, and a deep dive into C/C++ source code sounded like an opportunity too good to pass up. Our goal for this thesis was to deepen our knowledge about vulnerabilities, C/C++, and machine learning.

We would like to thank our supervisor, Donn Morrison, for all the help and guidance. Furthermore, we would like to thank NTNU for providing all the necessary resources. Thanks to Jan Grønsberg for all the support on the servers. Last, but not least, we would like to thank our families for their encouragement throughout this semester. Thank you all for your continuous support.

June 13, 2022, Trondheim

Mathias Knutsen and Eivind Hestnes Lervik

Contents

1 Introduction **1**
1.1 Background . . . 1
1.1.1 What are Bugs and Vulnerabilities? . . . 1
1.1.2 Traditional Analysis and Machine Learning . . . 1
1.1.3 Security Relations . . . 2
1.2 Research Questions and Goal . . . 2
1.3 Research Methods . . . 3
1.4 Contributions . . . 3
1.5 Thesis Structure . . . 3

2 Theory **5**
2.1 C/C++ . . . 5
2.2 C/C++ Memory Exploitation . . . 5
2.3 Common Weakness Enumeration . . . 6
2.4 Natural Language Processing . . . 7
2.5 Text Pre-processing and Code Representations . . . 8
2.5.1 Statistical and Trivial Characteristics . . . 8
2.5.2 Bag-of-Words and N-grams . . . 8
2.5.3 Token-Based Representations . . . 9
2.5.4 Tree and Graph-Based Representations . . . 9
2.6 Machine Learning . . . 10
2.6.1 Features and Datasets . . . 10
2.6.2 Artificial Neural Networks . . . 12
Gradient Descent and Backpropagation . . . 13
Convolutional Neural Networks . . . 15
2.6.3 Decision Trees . . . 16
2.6.4 Random Forest . . . 17
2.6.5 Evaluation . . . 17
Accuracy . . . 18
Precision and Recall . . . 18
F-score . . . 19
MCC . . . 19
ROC AUC . . . 19
PR AUC . . . 19

3 Related Works **21**
3.1 Introduction . . . 21
3.2 Annotated Literature . . . 21
3.3 Summary . . . 25

4 Methodology **27**
4.1 Data . . . 27
4.1.1 CWE-119 . . . 28
4.1.2 CWE-120/121/122 . . . 28
4.1.3 CWE-469 . . . 29
4.1.4 CWE-476 . . . 29
4.1.5 CWE-Other . . . 29
4.2 Pre-processing . . . 30
4.3 Lexing and Vectorizing . . . 30
4.4 Machine Learning Approaches . . . 32
4.4.1 Trivial Features using Random Forest . . . 32
4.4.2 BoW and N-grams using Random Forest . . . 32
4.4.3 Convolutional Neural Network . . . 33
4.4.4 Deep Representation Learning using Random Forest . . . 35
4.4.5 Other Combinations . . . 35
4.5 Experimental Setup . . . 35
4.5.1 Resources . . . 35
4.5.2 Libraries and Software . . . 36

5 Results **37**
5.1 Trivial Features using Random Forest . . . 37
5.2 BoW and N-gram using Random Forest . . . 37
5.3 Convolutional Neural Network . . . 39
5.4 Deep Representation Learning using Random Forest . . . 40
5.5 Result Summary . . . 42

6 Discussion **44**
6.1 Models . . . 44
6.2 Dataset . . . 47
6.3 Comparison to Literature . . . 47
6.4 Outstanding Challenges . . . 49
6.4.1 Lack of Data . . . 49
6.4.2 Feature Engineering . . . 50

7 Conclusion and Further Work **51**
7.1 Conclusion . . . 51
7.2 Future Work . . . 52
7.2.1 Better Data Collection . . . 52
7.2.2 Incorporate Tree/Graph-based Methods . . . 52
7.2.3 Detect First Then Classify . . . 52

References **53**

Appendices **56**
1. List of Manually Selected Tokens . . . 56
2. All Evaluation Metrics for RF-CNN-3 . . . 56

List of Source Codes

1 Hello World in C 5
2 Hello World in C++ 6
3 Example of a simple buffer overflow exploit in C 7
4 Function with repetitive identifiers 9
5 CWE-119 example 28
6 CWE-120 example 29
7 CWE-469 example 29
8 CWE-476 example 29
9 Custom Clang pre-processor input and output 31

List of Figures

1 ANN with two hidden layers 12
2 Illustrating CNN for sentence classification [14] 16
3 Visualization of the 100 most important 3-grams scored by the chi-squared-test 39
4 PR curves for RF-CNN-3 at 100 and 2500 estimators 41
5 PR curves for selected binary RF models (100 estimators) 43
6 ROC curve for selected binary RF models (100 estimators) 43
7 PR AUC for best multi-label model layered over results by R. Russell et al. [9] 48

List of Tables

1 Confusion matrix for predicted labels and true labels 18
2 VDISC Dataset - Distribution of vulnerable functions 28
3 Summary of CNN model 34
4 Metrics for RF multi-label and binary models trained using trivial features 37
5 N-gram combinations and number of extracted n-grams 38
6 Metrics for all RF multi-label models trained using n-grams 38
7 Metrics for all RF binary models trained using n-grams 38
8 Metrics for CNN-1 and CNN-2 with embedding dimensions 13 and 64 39
9 Metrics for all RF-CNN multi-label models 40
10 Metrics for all RF-CNN Binary models 40
11 CWE-119 Confusion Matrix for best model (multi-label) 41
12 CWE-120 Confusion Matrix for best model (multi-label) 41
13 CWE-469 Confusion Matrix for best model (multi-label) 42
14 CWE-476 Confusion Matrix for best model (multi-label) 42
15 CWE-other Confusion Matrix for best model (multi-label) 42
16 Confusion Matrix for best model (binary) 42

17 Our best models compared to literature. Weighted averages are used. 49

Acronyms

ADAM Adaptive Moment Estimation
ANN Artificial Neural Network
ANSI American National Standards Institute
API Application Programming Interface
AST Abstract Syntax Tree
AUC Area Under Curve
BCE Binary Cross Entropy
BLSTM Bidirectional Long Short-Term Memory
BoW Bag-of-Words
CFG Control Flow Graph
CNN Convolutional Neural Network
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
CWE Common Weakness Enumeration
ET Extra Trees
FN False Negative
FP False Positive
GNN Graph Neural Network
GPU Graphics Processing Unit
MCC Matthews Correlation Coefficient
ML Machine Learning
NLP Natural Language Processing
NVD National Vulnerability Database
PDG Program Dependence Graph
PR Precision-Recall
RAM Random Access Memory
RF Random Forest
RNN Recurrent Neural Network
ROC Receiver Operating Characteristic
ReLU Rectified Linear Unit
SARD Software Assurance Reference Dataset
SVM Support Vector Machine
TN True Negative
TP True Positive
VFG Value Flow Graph

1 Introduction[1]

1.1 Background

1.1.1 What are Bugs and Vulnerabilities?

A software bug can be defined as the underlying cause of a fault or unintended output from a program, usually originating from the source code. Common bugs can cause a program to terminate, cause the displaying of a wrong result, or have seemingly no effect at all. Bugs are common and are found even in large-scale software like Microsoft Windows and Linux kernels. To put this into perspective, Microsoft reported an average of 30 thousand bug reports produced each month across their development platforms [1]. A bug in itself is often harmless and has minimal effect on the end-user, but this is not always the case. More serious bugs can affect the software and render it useless or even worse; it can be exploited by an attacker. A bug exploitable by an attacker is also known as a vulnerability. Serious vulnerabilities, especially in large-scale software like operating systems and web browsers, can be exploited by attackers to run malicious code or grant unauthorized remote access. This is a serious security risk and should be addressed as soon as possible. Fixing a bug in production can often be many times more expensive than fixing it during development. Because of these factors, one wishes to reduce the number of bugs that passes quality assurance by using different measures. Catching bugs early in the development pipeline will reduce time and money spent down the line.

1.1.2 Traditional Analysis and Machine Learning

Today there are a lot of measures taken to avoid and prevent bugs in code. Testing and reviewing code are some of the most used methods, but as long as humans are writing code there, there will be room for error. Using static analysis tools, like those included in Visual Studio [2] and Clang [3], is also common, but does not guarantee bug-free code. Going one step further, dynamic analysis like fuzzing can be used. This is a rather time-consuming and heavy task to run, especially for large-scale projects. This is why this thesis will look at the state-of-the-art and proposed frontier solutions in this field that uses machine learning and static analysis to detect bugs. Machine learning is rapidly getting more popular and is now used in different fields all over the world. For tasks such as pattern recognition and anomaly detection, machine learning models can be faster and more effective than humans. One can not expect machine learning to perfectly detect all vulnerabilities and replace the prior proposed measures, but it can possibly further optimize the process. This thesis will look at different methods used to detect vulnerabilities in code using machine learning and what kind of results they have achieved. Different methods take very different approaches to solve the problem, where some rely on statistical

[1] Parts of this thesis consists of content from the specialization project (TDT4501) delivered autumn 2021.

analysis and others rely on the semantic relationships in the source code. This thesis will cover the biggest challenges in the field and potential solutions.

1.1.3 Security Relations

Bugs can range from harmless errors to vulnerabilities exploitable by a malicious attacker. The latter is the main topic of this thesis. The focus will be on vulnerabilities related to security and therein vulnerabilities related to low-level memory manipulation in languages like C or C++. These types of vulnerabilities are some of the most common and harmful, yet very difficult to detect. Open-source applications are known for being more secure in the sense that a lot of people are able to review and contribute to fixing bugs, but this is not always the truth. In April 2021, researchers at the University of Minnesota were caught having submitted multiple intentional vulnerabilities to the open-source Linux kernel [4]. The fact that events like these occur shows the clear lack of proper solutions to detect these kinds of vulnerabilities automatically.

1.2 Research Questions and Goal

The goal of this thesis is to research how to accurately and efficiently classify and detect vulnerabilities in C/C++ functions using machine learning. This thesis aims to improve upon existing approaches to detect and classify vulnerabilities using only the function source code. Unlike static analyzers like Clang, this research will only look at the static text semantics and ignore all references and context that exist outside the function scope. This approach will likely limit the overall performance, however, it has the advantage of being a more simple and lightweight problem that can be processed on most modern machines. Literature branching outside text semantic analysis will also be reviewed in order to compare results.

Research Question 1

How do common natural language processing approaches (statistical, n-grams, token-based) to vulnerability detection compare on a dataset of C/C++ functions?

Research Question 2

How do hybrid approaches combining deep representation learning and statistical features perform for vulnerability detection in C/C++ functions?

Research Question 3

Which combinations of tokens in C/C++ source-code prove to be important in order to successfully classify vulnerabilities?

1.3 Research Methods

The methods used to perform the research in this thesis were both exploration and experiment based. Literature on the topic of vulnerability detection using machine learning was first explored. A review of relevant literature was then produced. The research questions and goals were at this point re-evaluated, and the thesis scope was narrowed to better fit the time restrictions. The methodologies described in the remaining relevant literature were then attempted reproduced. The successfully reproduced models were used as a baseline while moving into the experiment phase. The experiments were based on more literature as well as our own experiences. Smaller goals were also set to make sure progress was made throughout.

1.4 Contributions

1. A review of literature on the topic of vulnerability detection and classification using machine learning.
2. An overview of how to implement vulnerability classification using machine learning approaches like:
   - 2.1. Convolutional neural networks
   - 2.2. Deep representation learning
   - 2.3. Random forest with n-grams and statistical features
3. Experiments and results using the aforementioned implementations
4. Source code for all the implementations can be found on GitHub at the following link: https://github.com/MathNuts/VulnerabilityClassification

1.5 Thesis Structure

1. **Introduction** An introduction to the topic of this thesis, as well as research questions and some background.
2. **Theory** Selected theory that supports the experiments and results of the thesis.
3. **Related Works** A look into the existing literature on the topic of vulnerability detection and classification for C/C++.
4. **Methodology** A description of the machine learning methods and respective datasets.
5. **Results** The results for the selected approaches.
6. **Discussion** Evaluation of the results, as well as a comparison of the results against previous literature.

7. **Conclusion and Future Work** A conclusion to the discussion and ideas for future work.

2 Theory

This chapter will cover the background theory necessary to understand the methods used in this thesis. This includes an introduction to C/C++, vulnerabilities, code representations and a select few topics on machine learning.

2.1 C/C++

C is a general-purpose programming language created in the 1970s. Despite its age, C is still one of the most used programming languages to this day. By design, C provides easy low-level memory access, as well as minimal run-time support enabling simple compiling over multiple platforms. The C language was standardized by ANSI in 1989, and the standard has been regularly updated ever since [5]. In 1979 C++ was made to extend C with new and modern features. Due to originally C++ being an extension, most of the C syntax remains usable within C++. The difference between the two is mostly new features like classes, inheritance, polymorphism, new keywords, and operators. It is worth noting that C++ does not make C redundant as C's inherent simplicity makes it a more efficient and better language for some demanding tasks, and for lower-end micro-controllers as it requires less run-time support. The syntax of both C and C++ contains keywords, identifiers, constants, strings, special symbols, and operators. The difference is in the amount of additional C++ elements. An example "Hello, World" program can be seen in Snippet 1 and Snippet 2 for C and C++ respectively. The syntax is very similar between the two, however, understanding the subtle differences is important when setting out to detect vulnerabilities. To put the popularity and usage of the languages into perspective we can look at some statistics. According to the 2021 Stack Overflow Developer survey, 21% of programmers use C, and 24% use C++ [6].

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, world\n");
}
```

Code Snippet 1: Hello World in C

2.2 C/C++ Memory Exploitation

Many vulnerabilities are related to reading from and writing to memory, usually by overflowing a buffer. To understand why this is a real threat, and why tools are needed to prevent them, one needs to dig a little deeper into how a program is executed. The example program in question can be seen in Code Snippet 3,

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Code Snippet 2: Hello World in C++

and is an example of a buffer overflow on the stack. In this example a user-type in a system has been set to "user" by default, however, exploiting the vulnerability, an attacker can change the type to fit their needs.

The name and type buffers are both allocated right after each other on the stack, and an attacker can therefore overwrite the second buffer by exploiting the *strcpy* function. Both the type and name buffers are 8 bytes long, so filling one buffer will require 7 characters in addition to the string terminator. To perform the exploit, simply write a name using 8 characters, and then the rest will overflow into the type buffer. As seen from the output of the program, the type has been successfully changed to "admin" by corrupting and overflowing the memory. For the sake of this example, the input string has been pre-defined in the program, but in a real-world scenario, the program will read from user input.

As one can see, these kinds of exploits can be dangerous, in this case even giving admin access to the exploiter. An attacker can also perform more advanced attacks replacing not only buffers on the stack but changing function calls, effectively allowing root access to the system. To counter these kinds of attacks safer function counterparts like *strncpy* have been added. These functions require the programmer to explicitly define the n first number of bytes of a buffer to read. However, even these functions are still exploitable unless the programmer chooses the right value for n.

2.3 Common Weakness Enumeration

To better help understand how to differentiate different vulnerabilities, one can refer to the common weakness enumeration (CWE) [7]. Different vulnerabilities will tend to have different traits, and differentiation between types is helpful for detecting these different types in the real world. For instance, a memory vulnerability will probably include specific functions and operations that are unique to reading and writing to a buffer. The common weakness enumeration, as its name suggests, is a list of known software and hardware weaknesses. It is sustained by a community project supported by large corporations as well as the U.S. government. The list consists of vulnerabilities of all types and for many different languages. The different CWE types are all described in detail, often with examples and references. This thesis will only look at CWE types related to C/C++.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    char type[8] = "user";
    char name[8];

    strcpy(name, "SOMENAMEadmin");

    printf("name:  %s\n", name);
    printf("type:  %s\n", type);
    printf("name addr: %p\n", (void *)name);
    printf("type addr: %p\n", (void *)type);

    /* Produces output:

    > name:  SOMENAMEadmin
    > type:  admin
    > name addr: 0x7fffbc83b658
    > type addr: 0x7fffbc83b660

    */
}
```

Code Snippet 3: Example of a simple buffer overflow exploit in C

2.4 Natural Language Processing

Natural language processing (NLP) is a field concerned with programming computers to process, analyze, and understand natural languages. Natural languages are very complex due to factors such as large vocabularies, semantics, syntax, and contextual nuances. Common tasks in the field of NLP include speech recognition, and interpretation and generation of natural languages among other things. A number of different techniques and algorithms are used in order to process and analyze natural languages. Programming languages are artificial, not natural languages, and are therefore quite different in a number of ways. However, programming languages, similar to natural languages, also rely on semantics and syntax. It should therefore be possible to process and analyze program code with NLP-based methods. In this thesis, the C/C++ program code will be processed by applying various NLP techniques.

2.5 Text Pre-processing and Code Representations

Before one can apply any machine learning method to detect vulnerabilities in source code, the source code needs to be represented as features that can be understood by a machine learning algorithm. Typically this involves converting the raw data into a vector of some sort. When converting to a vector it is important to understand the concept of granularity. The extracted vector should contain the optimal subsection of the original data. When detecting vulnerabilities this could be anything from class-level, to functions or lines. The appropriate granularity decides what ways one should represent the code. The three specific approaches to representation that will be covered are statistical characteristics, token-based, and tree/graph-based. The different approaches all have their own pros and cons, and the use cases can vary.

2.5.1 Statistical and Trivial Characteristics

The simplest way to represent any code fragment is to do a statistical analysis treating the code as text. This is easy to do and can capture various traits of the source code. Some possible features to extract are the size of the text, cyclomatic complexity: the number of possible paths through a piece of code, total counts of specific characters or words appearing in the text, character diversity: the number of unique characters in the text, and entropy. The processing of these calculations is simple, but they all have the same weakness, namely the lack of semantic and syntactic understanding. The order of the statements has no effect on the end result.

2.5.2 Bag-of-Words and N-grams

Bag-of-words (BoW) is a statistical way to vectorize a piece of text and is commonly employed in natural language processing. Each number in the BoW-vector corresponds to a word in the vocabulary, and the value represents the number of times that word occurred in the text. The vocabulary encompasses all the different terms that are counted, meaning that the final output vector will have a dimensionality equal to the vocabulary size. Usually, the vocabulary consists of all the unique terms that appear in all of the analyzed texts, however, a specified vocabulary can also be supplied. In the latter case, terms that do not appear in the vocabulary are simply ignored. When analyzing natural languages, it is common to ignore terms that are unlikely to contain useful information such as stopwords and punctuation characters. Consider the code in Snippet 4. In a basic BoW vectorizer, only considering words for the sake of the example, the resulting vocabulary would be $\{void, test, int, x, y, printf, d\}$. The resulting vectorized source code would then be $\{1, 1, 2, 2, 2, 2, 2\}$. Notice how both lists are of equal length. This is because the indexes of each list should match. When vectorizing actual code, it is common to include special characters as they are an essential part of code languages.

A BoW approach contains the frequency of each term in the text, however, it does not contain any information about the order the terms appear relative

```
void test(int x, int y) {
    printf("%d", x);
    printf("%d", y);
}
```

Code Snippet 4: Function with repetitive identifiers

to each other. In order to capture some information about the order of the terms, the BoW-model can be extended by using n-grams. N-grams are n long sequences of terms. The n-gram model is the same as the standard BoW-model, except it contains frequencies of each of the n-grams, rather than just singular terms. This has the advantage of retaining some order information from the original text, although if the size of n is large, the number of n-grams can be huge, especially if the vocabulary is large. A BoW-model is used to transform texts of arbitrary size into fixed-length feature vectors which can be used in machine learning models.

2.5.3 Token-Based Representations

Another simple representation can be achieved by performing a lexical analysis of the source code. By doing lexical analysis, one transforms the source code into a sequence of tokens. The sequence of tokens can be all the individual words, characters, op-codes or categories derived from the different words. The sequence of tokens is then usually converted into a vector representation. Word2Vec is such an architecture for vectorizing tokens [8] and is used in several works [9] [10] [11]. Word2Vec is a term encapsulating algorithms that can learn word embeddings from data sets, usually with the help of neural networks. A token-based representation is also often used in deep representation learning. Deep representation learning, also called feature learning, is a method used to replace manual feature engineering. This approach uses machine learning as an intermediate step to learn good features. These new features are then fed through a classifier at the end.

2.5.4 Tree and Graph-Based Representations

Trees are abstract data types consisting of hierarchically structured sets of nodes. A node can be linked together with other nodes as either a parent or a child. The node at the very top is called the root. Similarly, a graph is an abstract data type consisting of nodes connected by edges that can either be undirected or directed. Various types of trees and graphs can be used to represent different aspects of source code such as code structure, control flow, program dependencies, and syntactic structure. This allows higher-level semantics or relationships in the code to be captured as features.

A structure that is often used to represent code is abstract syntax trees. An abstract syntax tree (AST) is, as the name suggests, a tree representation of

the abstract syntactic structure of a piece of source code. The syntax tree is abstract because it contains the structural details or operations of the code. It does not contain inessential details such as parentheses and semicolons from the original text. This separates an AST from what is called a parse tree, which contains the whole code syntax. Each node in the AST represents a specific construct from the underlying code. The branches from each node as well as their location within the tree describe how the various constructs are connected. Code representation using ASTs has been extensively used [12] [11].

Control-flow graphs (CFG) are representations of all the possible paths through a program during execution. The nodes in the CFG represent a straight-line piece of code with no branches, also known as a basic block. The nodes in the CFG are connected by directed edges that illustrate how the program can execute. Features based on CFG representation of the code have been tried [10]. Another type of flow graph is the value flow graph (VFG). VFG syntactically represents semantic equivalence, with the nodes representing equivalence classes and the edges representing data flow.

A program dependence graph (PDG) is a graph representation of the dependencies within the code. Each node in the graph represents a statement or predicate. The edges represent the dependencies between the nodes. PDGs have also been used as an intermediate representation for generating features [13].

2.6 Machine Learning

Machine learning is a part of artificial intelligence and is the field concerned with creating methods that can learn. This learning is done by leveraging data to create a model that can perform tasks that it is not explicitly programmed to do. Machine learning is a large field with a wide range of different approaches such as supervised learning, unsupervised learning, and reinforcement learning. In this thesis only supervised learning is utilized. Supervised learning utilizes a set of labeled data, referred to as the training data, to create a model that can predict the label of new, unseen data. In essence, the goal of supervised learning is to use historical observations to predict future ones. Provided that historical labeled data is available, supervised machine learning can be used with many different types of data, for example images or text. As a result, supervised machine learning has potential applications in a number of different fields including medicine, finance, and marketing. Most machine learning models have a set of tunable parameters that can be adjusted in order to improve the performance of the model. Such parameters are referred to as hyperparameters.

2.6.1 Features and Datasets

Good quality data is paramount in supervised machine learning. To create a supervised model, data samples with labels are needed. The label dictates which class each data sample belongs to. Often there are only two possible classes for data to belong to, in which case it is referred to as binary classification, but

it is not uncommon to have more classes. There is also a distinction between multi-class classification, where each sample belongs to only one of the classes, and multi-label classification, where it is possible for individual data samples to belong to multiple different classes. Some examples of class labels are whether an image contains a specific entity, whether the sentiment of a text is positive or negative, or whether a piece of source code contains a bug. How a set of data has been labeled varies depending on the type of data, but it is common that the data has to be labeled by human experts. The information that makes up the data itself is referred to as the features. Features are a set of variables that, ideally, are correlated with the class label. Features can be numeric, binary, or categorical (nominal or ordinal) and are derived from the raw data. Feature extraction is the process of extracting features from raw data. Features vary in quality and some features will be better than others. Good quality features are discriminatory with regards to the class label, and thus good at separating the data into the different classes. Features can be ranked by their correlation to the class labels, and lower-quality features can be removed through the process of feature selection. One possible way to rank features is to use the chi-squared test for each feature with regard to the class label. This is a test to check whether two stochastic variables are independent. The chi-squared test produces a score based on the null hypothesis that two variables are independent. A low score indicates that there is a high probability that the variables are independent, whereas a higher score indicates that the variables are dependent. When selecting features, it is desirable to have features that are dependent on the class label, thus features with a high score are kept. The process of feature selection reduces the dimensionality of the data. The dimensionality refers to the number of features that are in a set of data samples. The complexity and processing time of a model increases with dimensionality, and as such, it is often beneficial to remove redundant features and reduce dimensionality.

The data samples used when creating a model are in the form of vectors, where one (or multiple in the case of multi-label data) element(s) is the class label(s), and the other elements are the features, each element representing a specific feature. Multiple data samples are generally referred to as a dataset. When building a model, it is common to split the dataset into two or three subsets: training, testing, and validation. The training set is used for creating or "training" the model and is usually the largest subset. The testing set is used to test the performance of the finished model. A validation set is sometimes used to evaluate the performance during the training phase in order to optimize the model. In some cases, it is possible for a machine learning model to "learn" too much of the specific intricacies of the training data, which results in poor performance when classifying new data. This behavior is known as overfitting. To avoid overfitting and create a more generalized model, one can iteratively change the train and test sets until all samples have appeared in both the training and testing data. This iterative process is referred to as cross-validation.

2.6.2 Artificial Neural Networks

Artificial neural networks (ANN) are a class of related algorithms used for machine learning, and specifically representation learning. Representation learning is a part of machine learning that deals with models that are able to automatically create high-level features from raw data. Artificial neural networks consist of a number of connected nodes, structured in layers. The nodes attempt to imitate biological neurons where synapses transmit signals between connected neurons. An illustration of the concept can be seen Figure 1. In essence, an artificial neural network is a function that, given an input, attempts to give the correct output.

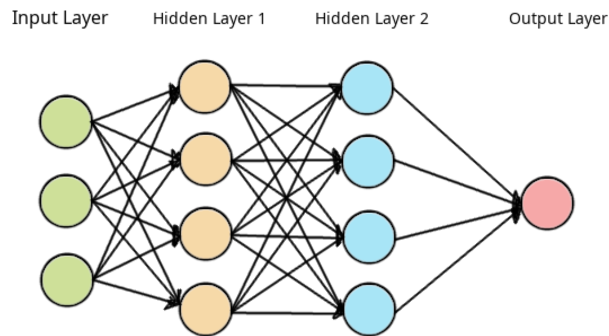


Figure 1: ANN with two hidden layers

There exist multiple different types of ANNs, but their mechanism of action is similar for the most part. A basic feed-forward, fully connected neural network will be explained here. As previously mentioned, the nodes are structured in different layers. The first layer is known as the input layer and has nodes equal to the dimensionality of the input data. The final layer is the output layer and has nodes corresponding to the number of classes. The node layers between the input and output layers are known as hidden layers. In a feed-forward network, information travels forward in one direction from the input layer to the output layer. In a fully-connected network, each node in a layer transmits its "signal" to every node in the following layer. Nodes are essentially functions that take input values from all nodes in the previous value and produce a specific output value based on the input. This value is the signal that a node transmits to the nodes in the next layer. When classifying, the network takes a fixed-length numerical vector as input. The values from the input vector are the values that the input layer feeds forward to the next layer. Each connection between nodes has a weight. The value passed from one node to the next is multiplied by this weight. A node in a hidden layer receives a value from every node in the previous layer, and each value is multiplied by the weight of its specific connection. The value of a node in the hidden layer thus becomes the weighted sum of each value from the previous layer. Additionally, a bias value is added to the sum. Equation 1 shows how the output of a node is computed, where x_j

is the indexed value from a node in the previous layer, w_{ij} is the weight on a specific connection between a node in the previous layer and the current node. The bias is represented by b.

$$z = b + \sum_{j=1}^{n} x_j w_{ij} \tag{1}$$

Before a node transmits its updated value to the nodes in the next layer, an activation function is applied to the value. The non-linear activation function adds non-linearity to the network. Additionally, an activation function can restrict the values to any desired range. There are multiple common activation functions that serve different purposes, and different layers in a network can have different activation functions. Some common activation functions are *sigmoid* and *ReLU*. The sigmoid function 2 squishes any input value to a value between 0 and 1.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

The rectified linear unit (ReLU) 3 converts negative values to 0 while retaining positive values.

$$ReLU(x) = max(0, x) \tag{3}$$

The nodes in all the hidden layers are all updated, layer by layer, each layer outputting a vector that is used in the next layer. The output vectors of the hidden layers can be considered as features that have automatically been extracted by the neural network. These features might not necessarily make sense to humans in the same way manually extracted features do, but they can still be good features.

The final, output layer of a neural network works slightly differently than the hidden layers. The output layer has dimensionality corresponding to the number of possible output classes, and the network's classification is the class corresponding to the output node with the largest value. A special activation function, such as *softmax*, is sometimes used before the final layer in order to get the output as a probability distribution between the possible classes. One additional thing to point out is that it is possible to use neural networks to extract features that can then be classified using a different model.

The idea of a neural network as a network of nodes transmitting data to each other is just a theoretical way to think about it. In reality, the model consists of vectors and matrices that are multiplied together. The output of a layer in the network can then be written as Equation 4, where a^n is the output vector of layer n, W is the weight matrix, b is the bias vector, and σ is the activation function.

$$a^n = \sigma(Wa^{n-1} + b) \tag{4}$$

Gradient Descent and Backpropagation

The previous section explained how data is classified by doing a forward pass through a neural network. The output of the neural network is dependent on

the values of all the different weights and biases. When talking about training a neural network, it actually refers to adjusting the values of the weights and biases in such a way that the network provides the correct output to a given input. Before training, all the weights and biases in the network will be randomly initialized, which means that the network will not perform well. To train it, a set of labeled training data is utilized. Training samples are fed into the network in order to receive the output vector that can be compared to the label. To compare the predicted output to the expected label, a loss function is used to compute the loss with respect to the weights of the model. The loss function says something about how the model performed. By training, the goal is to reduce the loss of the model. Binary cross-entropy 5 is a commonly used loss function. In this loss function y_i is the target value, and p_i is the predicted value.

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i) \tag{5}$$

The negative gradient of a multi-variable function provides the direction where the function decreases the fastest. The gradient descent algorithm uses the negative gradient to reach a local minimum of a function. It does this in an iterative manner, by finding which direction to "move" the weights in order to reduce the cost the fastest, then "moving" an amount in this direction, repeating the process until a local minimum is reached. The updated weights, w^t, can be computed using the current weights by the gradient descent formula 6. The gradient vector $\nabla C(w^t)$ reveals what changes should be done to each weight, i.e. which weights should increase, which should decrease, and the magnitudes of each weight's effect on the loss. A scalar factor, γ, is used to control how much the weights should be updated, and can be thought of as step size when moving in the gradient direction. This factor is referred to as the learning rate and is a hyperparameter. A small learning rate will require more steps to converge, while a large learning rate can potentially miss local minima by overshooting. It is possible to use an adaptive learning rate that changes during training. To compute the gradient of the loss function, an algorithm called backpropagation is used. The gradient is found by computing the partial derivatives of each variable, and backpropagation works by using the chain rule to compute the gradient of the loss function for each layer in the network, working backward recursively from the last layer, one layer at a time. Using backpropagation to compute the gradients each time, gradient descent can be used to iteratively update the weights and biases until a local minimum for the cost function is reached.

$$w^{t+1} = w^t - \gamma \nabla C(w^t) \tag{6}$$

How the weights should be adjusted to best decrease the loss function over the whole training set, can be found by finding the average of all the gradients from all the training samples for each step. However, computing the gradients for every single sample for each weight update is very computationally heavy. Instead, training batches can be used. Batches are small subsets of the total

training data that are fed to the network, and the average gradient of each batch, rather than the whole training set, is used for each step. The average gradient of the batch approximates the average gradient for the entire training set while requiring fewer computations per weight update. The batch size is a hyperparameter and is the number of samples that are fed through the network before the weights are updated. To optimize the learning process further, it is common to use an optimization algorithm such as adaptive moment estimation (ADAM). It is probable that the model weights are not optimal after all the training samples have been passed through the network once. It can therefore be beneficial to pass the training set through the network more than once. Each time the whole training set is passed through the network is referred to as an epoch. The number of epochs is a hyperparameter. Similar to other machine learning models, neural networks can also be susceptible to overfitting. One way to prevent overfitting is to use dropout. Dropout is a technique where random weights in the network will be dropped by setting their value to 0. This is done to make the model more generalized, and by preventing very specific features from the training set to be learned. The probability of a weight being dropped is a possible hyperparameter.

Convolutional Neural Networks

Convolutional neural networks are a distinct type of neural network that is commonly used for tasks involving images as input data. They are named for their special convolutional layers. These layers use filters, or kernels, to perform a convolution operation on the input. This involves having a kernel matrix move across the input and computing the dot product between the kernel and the underlying input for each step. This is done until the whole input has been processed. Each of the computed dot products is stored in an output matrix. These outputs are referred to as feature maps. It is common to create several feature maps, using different kernels, for each input sample. The feature maps are then combined and fed forward to the following layers in the network. The values in the kernels are randomly initialized but are updated when training the network. Since the kernel values are weights similar to any other weight in the network, they are also updated the same way using gradient descent and backpropagation. Pooling is an approach to reducing the size of the feature maps. Max pooling is a common pooling technique that involves keeping only the maximum value in a given interval while discarding the rest. As a result of the convolution operations, convolutional neural networks have a few additional hyperparameters. Stride refers to how much a kernel is moved between each step when moving across the input. The size of the kernel matrices can also vary, as well as the number of kernels to use.

While very often used for image classification purposes, CNNs can be applied to many problems, and a popular scenario is sentence classification. This application of a CNN was first proposed by Yoon Kim in 2014 [14]. The basic idea behind this method is to convert each word in a sentence into a fixed-size vector resulting in a $n \times d$ matrix, where n is the number of words and d is the

number of dimensions. This fixed vector, also called a word embedding, is either a result of randomized numbers or a pre-trained vector using Word2Vec or similar architectures. From this point, convolutional filters of size $x \times d$, where x is the filter length in words, are applied in parallel. All the resulting vectors are then max-pooled along their axis in order to output only a single value. The single value for each filter is then appended to the resulting vector. If two convolutional filters are used making 512 feature maps each, the resulting vector will be 1024. The resulting vector is then fed through linear layers to predict the final class. An illustration of this process can be seen in Figure 2. This method was originally made by Yoon Kim in order to classify natural language, however, the approach can be applied to many other texts including code. As natural languages and code are very different in structure and vocabulary size, the parameters used may have to be tweaked.

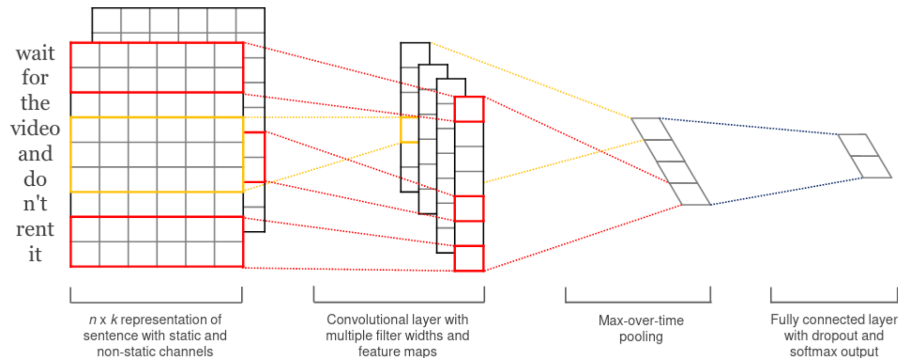


Figure 2: Illustrating CNN for sentence classification [14]

2.6.3 Decision Trees

Decision trees are widely used in machine learning, and are most often used within other methods rather than by themselves. Decision trees are used both for regression when predicting a real number, and classification when predicting discrete labels. The decision tree consists of connected nodes, each internal node representing a specific feature in the data, and the edges representing specific values for that feature. Leaf nodes correspond to a specific class label. When classifying, start at the root node of the tree, and work down the tree by moving along the edges corresponding to the feature values of the sample that is being classified. When a leaf node is reached, the sample is classified as the class label corresponding to that leaf node. There are several different variations of algorithms for creating decision trees, but in general decision trees are created as follows: Start by splitting the supplied training data into smaller subsets. The data is split by looking at which of the features in the training set can best separate the data into the different target labels. To calculate which feature is

most suitable, a gain function is used. Commonly used gain functions are gini impurity 7 and information gain using entropy 8. The same process is applied recursively each time on each created subset, until all the samples in a subset possess the same class label, or splitting no longer yields any additional value.

$$I_G(p) = 1 - \sum_{i=1}^{J} p_i^2 \tag{7}$$

$$H(T) = - \sum_{i=1}^{J} p_i log_2 p_i \tag{8}$$

2.6.4 Random Forest

Random forest is a supervised ensemble machine learning method that can be used for classification or regression. Ensemble learning methods are machine learning methods that combine multiple diverse models. In a classification problem, each model in the ensemble provides its own classification, and the ensemble's prediction is the majority result from all the individual models' predictions. When performing a regression task, the average or mean of the predictions is used. Random forest consists of an ensemble of decision tree models. Each decision tree in the ensemble is created using a random subset of the training data, and sometimes also using only a subset of the features. This overcomes the issue of overfitting on the training data that a single decision tree can suffer from. As a result, random forests usually outperform decision trees. The number of decision trees, sometimes called estimators, used in a random forest model is a hyperparameter and can greatly affect performance, with more trees generally performing better, up to a certain point.

2.6.5 Evaluation

To rate the performance of a machine learning model, a range of evaluation metrics can be used. These metrics are not only useful to determine if a completed model is performing well or not, but it is also an essential part of training and adjusting models. The metrics that will be covered in this thesis are strictly related to classification as this thesis is about vulnerability classification and detection. The reason to use multiple metrics is that they all reflect different aspects of the model. For one problem one might prefer to have high specificity, while other times only the sensitivity matters. The different metrics used in this paper include accuracy, precision, recall, ROC (receiver operating characteristic) curves, PR (precision-recall) curves, F-score, and MCC (Matthews correlation coefficient). For ROC and PR, the area under curve (AUC) is used to convert the curve's information into a single-number metric.

All the aforementioned metrics are in some way or another related to the predicted label compared to the true label of a model. The four resulting cases in classification can be seen in Table 1. The two true cases are true negative (TN), and true positive (TP). The true negative represents a negative sample that

has successfully been predicted as negative, while the true positive represents a true sample predicted as true. A false positive (FP) occurs when a sample is incorrectly predicted as positive, while a false negative (FN) occurs when a sample is labeled negative when it is positive. Understanding these four cases is crucial in order to understand the meaning behind the other metrics. In addition to these four cases, it is also important to understand that a machine learning model returns a float between 0 and 1, and not a true or false. This float is automatically changed into true/false values using a threshold of value 0.5. Some metrics use custom thresholds to evaluate the model.

		Predicted Label	
		Negative	Positive
True Label	Negative	TN	FP
	Positive	FN	TP

Table 1: Confusion matrix for predicted labels and true labels

In a multi-class scenario, the metrics will have to be calculated for each class, and then the scores are merged. This is usually done either by averaging over the classes or by doing a weighted average so that all classes are given the same weight disregarding the number of samples in the test set. Another solution is to skip the merging step and look at each class' score individually.

Accuracy

Accuracy is the most simple and commonly used metric. Accuracy tells us how many true predictions were made compared to the total number of entries in the dataset. The formula for calculating accuracy is given as follows:

$$accuracy = \frac{TN + TP}{TN + FP + FN + TP} \tag{9}$$

For a balanced dataset, this measure functions well, however, on a dataset consisting of 90% TP samples, the accuracy will be at 90% even if all predictions are positive and none of the negative samples are predicted. To counter this, precision and recall can be used in their stead.

Precision and Recall

Precision and recall are two closely related metrics in model evaluation. While accuracy only looks at true predictions over the whole dataset, precision and recall look at true labels versus the false labels. Precision is defined as follows:

$$precision = \frac{TP}{FP + TP} \tag{10}$$

This precision value, in other words, is the fraction of TP over all positive predictions. Recall, also called sensitivity, is defined as follows:

$$recall = \frac{TP}{TP + FN} \quad (11)$$

This recall value is the fraction of TP over all true labels. These two values can in combination tell us a lot about the model's ability to detect true samples.

F-score

The F-score, also called F1-score, is one metric that uses precision and recall to calculate an average score that measures a model's classification accuracy. The highest value of this score is 1, and the lowest is 0. The F1 score is calculated as follows:

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} \quad (12)$$

MCC

The MCC score is a rather peculiar score and is not very commonly used compared to precision, recall, and F1. It is however used in cases where the data is severely imbalanced, as it handles this very well [15]. The MCC score handles imbalanced datasets by ensuring good results in all four prediction values (TN, FP, FN, and TP). The score ranges between -1 and +1, where 0 is no correlation, +1 is a positive correlation and -1 is a negative correlation. The MCC score is defined as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (13)$$

ROC AUC

ROC AUC is one of the more popular metrics as it captures the model performance really well. However, this is not always the case for imbalanced datasets. ROC AUC is the area under the curve of the receiver operating characteristic (ROC) curve. The ROC curve uses the FP rate and TP rate on its x and y-axis respectively. The FP rate or FPR and TPR are defined as FP over negatives and TP over negatives respectively. Machine learning models predict a value between 0 and 1, and adjusting the threshold can change the resulting class. FPR and TPR are calculated for a number of thresholds between 0 and 1. The resulting ROC graph then tells us about what TPR one can achieve with all thresholds of FPR. This is useful when manually choosing a threshold for the final model. Note that this is the ROC curve by itself. The final score is the area under curve (AUC), resulting in the ROC AUC score.

PR AUC

PR AUC, also called mean average precision (mAP), is another metric similar to the ROC AUC, however, the x and y-axis are the recall and precision values

respectively over thresholds between 0 and 1. This metric is much better at handling imbalance in data than ROC. In this curve, one can see at what levels of recall one achieves the wanted level of precision. This is also a great indicator of where to put the final threshold. The PR AUC score is calculated by taking the area under the curve.

3 Related Works

3.1 Introduction

This chapter will explore some of the published literature on the topic of bug and vulnerability detection using machine learning. The primary focus is on works that deal with detecting security vulnerabilities from source code using machine learning, however, related literature on bug and vulnerability detection was also researched in order to get a more comprehensive overview of the topic. Methods for extracting features from the raw data were the primary concern when researching literature, rather than which classification methods were used. The literature search provided the starting points for the implementations that will be covered in the later chapters of this thesis.

Below is a summary of some of the articles that were found during the literature search. These were selected because they demonstrated interesting and relevant approaches and promising results. For each article there is a short paragraph, summarizing the methodology and results of the article, as well as briefly commenting on possible drawbacks of the article. Due to the differences in datasets utilized, the complexity of the data, and the nature of the problem itself, it is difficult to draw any exact conclusions in regards to which methodology is best simply by comparing the different works. It does, however, provide useful insight into different methodologies, possible problems, and what has worked well for others.

3.2 Annotated Literature

Vulnerability detection using n-grams and statistical characteristics

Chernis et al. [16] demonstrate the possibility of catching a large percentage of bugs using machine learning classifiers on text features extracted from C source code. They have taken two different approaches to this problem, the first of which was to extract simple statistical characteristics like character count, diversity, entropy, nesting depth and different word counts. The second approach was to use more complex, yet statistical characteristics like n-grams and suffix trees. The first data set they compiled consisted of 100 vulnerable functions extracted from Github, as well as 100 non-vulnerable functions from the same files. The second data set contained the same vulnerable samples, but non-vulnerable functions from popular Linux programs that are unlikely to contain bugs. For the first data set, the simple features turned out to be better performing than the more complex ones. The simple features all exceeded 60% percent accuracy and character diversity alone reached 75% using a K-means classifier. The n-gram and suffix tree approach reached a maximum accuracy of 63.5% and 60% respectively using a naive Bayes classifier. Cross-validation was used in all three cases. As for the second data set, a combination of simple features and n-grams performed the best at 69%. The data used in this approach is scarce and a conclusion as to what works best is difficult to make based on this fact. However, the results do suggest that even simple statistical characteristics

can give us some information about whether a function contains a bug or not. Combining simple features with more complex ones like syntax trees or natural language processing might give satisfactory results.

Bug detection by introducing soundness to static analysis tools

Heo et al. [17] take a unique approach to the problem by introducing soundness in already existing unsound static analysis tools. The many unsound static analysis tools that are used today, all miss a large portion of true positives (TPs) in exchange for having a small number of false positives (FPs). Sound analysis tools are often heavy to run and introduce huge amounts of FPs. This approach introduces some soundness to detect these missed bugs, as well as keeping FPs down to a minimum. To do this they selectively apply soundness to functions that are likely to contain bugs. To determine which functions probably contain bugs, they use a One-Class SVM classifier. This is an unsupervised algorithm based on SVM that detects outliers. The data set used consisted only of harmless code, as the classifier only requires one class to determine outliers. The features they use for this classifier are 37 handcrafted semantic and statistical features. As for the evaluation, a data set of 23 open-source C programs were used, containing a total of 138 documented bugs. For reference, applying unsound analysis resulted in 33 TP and 104 FPs, and using sound analysis resulted in 118 TP and 677 FPs. Using the proposed hybrid method they achieved 100 TPs and 264 FPs. The FP rate is still considerably high compared to the unsound analysis, but in exchange, they get close to the TP rate of true sound analysis with less than half the FPs.

Bug detection using n-grams and op-codes

Chappel et al. [18] from Oracle labs apply various off-the-shelf machine learning techniques for detecting bugs in C programs and compare the results to other static bug detection tools. The different techniques are trained and tested on both publicly available data, as well as some of Oracle's own internal code bases. The data used includes artificial examples of just a few lines of code that illustrate the presence or absence of certain types of bugs, as well as real code examples where specific bugs have been reported. For some of the data used, the bugs have been reported by Oracle's internal static bug detection, and then manually evaluated to create a ground truth. In this data, not all bugs present in the code are necessarily labeled, since only bugs found by the tool are evaluated. Several types of features are considered. The first type relates to n-grams of instruction opcodes (load, store, etc.), and counts how many times a sequence of n specific opcodes appear in a function. The second type of features were complexity measures such as lines of code, nesting depth, and cyclomatic complexity. Various text features from the source code such as reserved words and parenthesis are also extracted. Over 2500 different features are extracted from the source code in total. Using a random forest classifier, models were trained to detect different types of bugs in the source code, with

a separate model used for each class of bugs. The trained models were able to match or even slightly outperform the static analysis tools on some classes of bugs but had fewer correct detections overall. They did, however, also have fewer false positives, being able to correctly identify 46 out of 116 bugs with 3 false positives compared to 56 correctly identified and 34 false positives by their static detection tool. However, the machine learning-based models only achieved adequate results when relying on features from the other static analysis tools.

Vulnerability detection using deep representation learning

Russell et al. [9] demonstrate how using deep representation learning is a promising approach for detecting bugs in source code. Their method directly interprets lexed source code and sends it through a CNN or RNN for the generation of new features. The data used is on function level and consists of functions from the SATA IV Juliet Test Suite (a benchmark for static and dynamic analyzers), Debian source code, and public Github repositories. The data is processed and run through a custom lexer that represents the source code with a vocabulary of 156 tokens. This lexer filters out comments and combines similar types of words to reduce detail and overfitting. To prevent other kinds of biases, a strict filter removed about 90% of the raw functions, as these consisted of real duplicates and duplicate lexed representations. Using a lexed function, they embed it into a $l \times k$ matrix, where l is the number of tokens and k is a fixed-length randomized vector for each unique token. This matrix is then used as the initial embedding for the CNN and RNN. Using the neural networks to perform the final classification turned out to be worse than using them for feature engineering, and they, therefore, used the final layers and states as features for a RF classifier. To evaluate the performance of their approach they also trained a RF classifier using a BoW representation that ignores the token order. As for the results, the CNN slightly outperformed the RNN both as a classifier and for feature generation. On the Juliet Test Suite, the CNN alone and CNN with RF achieved a ROC AUC of 0.954 and 0.936 respectively. The BoW baseline achieved a lower score of 0.913, indicating that their approach is indeed benefiting from being able to retain the order of tokens. For the combined Debian and Github data set, the highest ROC AUC achieved was with the CNN and RF at 0.904. In this case, the baseline was a little closer at 0.883, but still quite a step down.

Vulnerability detection using abstract syntax trees

Bilgin et al. [12] use an AST representation and a CNN in order to attempt to distinguish vulnerable and non-vulnerable software at a function level. Before they could feed the AST to the neural network they need to convert it to a numeric vector. In order to preserve the structural relations of the AST as a one-dimensional vector, they converted it to a binary tree first. In a binary tree, all internal nodes have two children, and they made sure that all leaf

nodes have the same depth. By performing this conversion, the tree will be of equal size for each function, with the same placement of nodes, and therefore be optimal input for a CNN. Empty nodes have a default NULL value. They then convert the binary tree into a vector sorted by depth. This vector now consists of tokens, and to do its final conversion, all equal tokens are converted to their own unique numerical three-dimensional vector. Increasing the depth of the binary tree resulted in an exponential increase in processing time, and for performance reasons, they chose a depth of 8 for the evaluation. The dataset used is an undersampled and balanced subset of the dataset proposed by Russell et al. [9]. The subset includes four different CWE types. The performance of the model varied according to what vulnerability was classified. CWE-476 (NULL Pointer Dereference) had an AUC ROC score of 0.882, while CWE-120 (Classic Buffer Overflow) got a lower score of 0.778. This study shows that some types of vulnerabilities can be significantly harder to detect than others.

Vulnerability detection using control flow graphs and BoW

Harer et al. [10] use two different types of features to create an automatic vulnerability detection system for C/C++ code at a function level. The first type are build-based features extracted during the compilation process. These features include the CFG of the code, as well as the opcode vectors and use-def matrices from the basic blocks. The second type are source-based features extracted directly from the code. This includes using a custom lexer that categorizes the code elements into different bins such as string literals, numbers, operators, keywords, function calls, etc. Two sets of vectorized features are extracted from the lexed tokens using a bag-of-words (BoW) approach and the word2vec algorithm respectively. Two distinct data sets are used for testing. The first is the full set of C/C++ packages distributed with Debian, while the second is a large set of C/C++ functions from public Github repositories. The data is then labeled by analyzing the code with the Clang static analyzer. The code is labeled as either "good" or "buggy", using the result from the static analyzer as the ground truth. Based on the fact that R. Russell contributed to this work as well, it can be assumed that the dataset is similar to the one described by R. Russell et al. Using the BoW vector and an extra-trees (ET) classifier, a ROC AUC of 0.85 was achieved. Using a CNN initialized with the word2vec representation achieved a slightly better result with a ROC AUC of 0.87. By using the features from the CNN-model as input in an ET-classifier, the precision-recall AUC increased slightly while ROC AUC remained at 0.87. Using the build-based features with a random forest classifier performed worse overall, achieving ROC AUC scores of 0.76 and 0.74 on the Debian and Github data sets respectively.

Vulnerability detection using program dependence graphs

Li et al. [13] investigate whether more semantic code information helps with detecting vulnerabilities related to specific library/API calls in source code. This

is done by comparing classification using just data dependency features to using control dependency and data dependency features at the same time. They use the open-source tool Joern [19] to analyze programs and generate various features. The program's AST is extracted and used to identify library/API function calls. Then, a program dependence graph is generated for each function. Program slices that show data dependency and control dependency are generated from the PDG. Finally, code gadgets are generated from the program slices. Code gadgets are lines of code in the order of the dependency between them. The code gadgets are vectorized by mapping to a sequence of symbols which are transformed into fixed-length vectors. After testing using code gadgets with just data dependency and comparing with also using control dependency, the authors conclude that using both control dependency and data dependency performs better than using just data dependency alone. A range of different neural networks was tested in order to decide which is better for vulnerability detection. A bidirectional long short-term memory (BLSTM) neural network is found to perform the best, achieving an F1 score of 92.4 on a data set of 68,353 code gadgets where 13,686 were labeled as vulnerable. Different methods to accommodate imbalanced data are tested, including oversampling based on interpolated data, as well as undersampling, however, both of these performed worse than using no sampling.

Vulnerability detection using graph neural networks

Cheng et al. [20] leverage the recent advancements in graph neural networks (GNN) to represent the source code while maintaining high-level programming logic. A PDG is constructed from the source code by combining the VFG and CFG. From this PDG they extract structured information, that can be directly used in the GNN, as well as unstructured information. The unstructured information needs to be filtered and vectorized using a word2vec approach before it can be used as input in the GNN. The GNN used in this paper is a k-GNN, a k-dimensional GNN, understanding higher-order graphs. The data used is collected from SARD, a vulnerability database containing a wide range of CWE categories. The final data set contained about 150k vulnerable samples and 400k safe samples after extracting samples from the source files. The final GNN network got an average F1 score of 0.956 over the different CWE types. They also compare their k-GNN approach with other GNNs and achieve very similar, but slightly worse results. Even though graph neural networks are very new, they seem to perform particularly well in classifying vulnerable snippets of code. The clear downside to this complex approach is the amount of context and processing needed to extract the different graphs.

3.3 Summary

In this chapter, a total of eight pieces of literature have been annotated. The type of machine learning used, as well as methods of pre-processing and representations are many. According to the literature, trivial features do not perform

very well compared to more advanced representations like lexed and graph-based representations. The majority of the approaches use graph-based representations of some sort, but they all have the same drawback; they require the surrounding code and context. The NLP-based approaches using a lexed representation only, do not have this problem, and they still achieve similar results.

As stated when introducing the goal and research questions, this thesis will focus on text semantics and not context captured by graph-based representations. However, it is still important to review the literature on different topics as their methodologies can be compared, and new ideas can come to life.

4 Methodology

In this chapter, a range of NLP approaches to vulnerability detection are described. Multiple methods of pre-processing have been applied to create different representations, all based on previous literature. The machine learning models are also inspired by the previous literature, however, new combinations of models and features are explored. The main approaches include the representation learning method used by R. Russell et al. [9], as well as n-grams as used in the works by B. Chernis et al. [16] and Y. Pang et al. [21]. All pre-processing and representations of data described in this section were calculated using only the train set before transforming the whole dataset.

4.1 Data

The Draper VDISC dataset is the main dataset used in this thesis. The dataset consists of 1.27 million functions labeled by their CWE number. It was created by R. Russell et al. [9] as a part of their approach to classifying vulnerable functions. The functions have not been labeled manually, nor by dynamic analysis, but rather by static analysis. The static analysis tools used for labeling were a combination of multiple open-source analyzers including Clang [3], Cppcheck [22], and Flawfinder [23]. R. Russell et al. used a team of security researchers to properly map each vulnerable result to its correlated CWE. This approach to labeling might introduce a number of incorrectly labeled samples, but other options like using pull requests were deemed too hard to automate, and it is too time-consuming to manually label large amounts of data. Even without a perfect dataset, this method should still prove sufficient to train models to mimic the behavior of static analysis. Other means of improving the dataset have been done. Strict data curation has already been performed on the data. This curation included the removal of all duplicates that could result in overfitting, as well as removing functions that compiled into the same set of op-codes. To put this into perspective, the 1.27 million entries in the dataset are only about 10% of the originally collected data samples.

The function sources are fetched from popular GitHub repositories, as well as the Debian Linux source code. Only the top four CWE types are labeled individually, while the rest are bundled into their own category. A function can have one or more CWEs at the same time. This dataset is very unbalanced, containing only about 6.5% vulnerable samples across all classes. This is however, a very real-world scenario compared to balanced datasets like the Juliet Test Suite [24]. The dataset is also pre-split into train, validate and test versions using an 80:10:10 split. This makes it easy to compare with results from other papers. An overview of the dataset can be seen in Table 2.

The Juliet Test Suite is also used in some of the experiments. This suite is a collection of test cases for C/C++. This suite was made by the National Institute of Standards and Technology (NIST) as a benchmark for static analysis tools. Similar to the VDISC dataset, this collection also labels the vulnerable samples by their CWE type, but does so by including 118 different types. An-

other difference is that this dataset is a collection of compilable files with helper functions. This makes this dataset more complex and not restricted to specific functions. The Juliet Suite consists of 64 099 test cases where each case consists of at least one vulnerable, and one safe function. All test cases in the Juliet Test Suite has been artificially created and does not represent real-world code, but rather fabricated vulnerable functions and their fixed counterpart.

In the next part of this chapter, five relevant categories of CWEs are described with examples. These five categories make up the VDISC labels, and they will be the main focus for classification. Understanding how the different CWE types are produced can help during lexing and pre-processing of the source code.

CWE	Train	Validate	Test
CWE-119	1.9%	1.9%	1.9%
CWE-120	3.7%	3.7%	3.8%
CWE-469	0.2%	0.2%	0.2%
CWE-476	1.0%	0.9%	0.9%
Other	2.7%	2.8%	2.7%

Table 2: VDISC Dataset - Distribution of vulnerable functions

4.1.1 CWE-119

CWE-119 is the second most common vulnerability in the VDISC dataset. It is defined as an improper restriction of operations within the bounds of a memory buffer [25]. Generally, this means that the programmer does not check whether or not the requested index is in the allocated buffer. An example of a typical CWE-119 vulnerability can be seen in Code Snippet 5.

```
int main(void) {
    char *items[] = {"one", "two", "three", "four"};
    int index = GetRandomIndex();
    // Unsafe - This index might be outside bounds!
    printf("You selected %s\n", items[index]);
}
```

Code Snippet 5: CWE-119 example

4.1.2 CWE-120/121/122

CWE-120, 121, and 122, are all vulnerabilities related to buffer overflows. CWE-120 is today often referred to as a "Classic Buffer Overflow", as it is the most common one. In this case, the programmer simply forgets to check the length of the input and ends up writing more data than the space they have allocated.

An example of this vulnerability can be seen in Code Snippet 6. CWE-121 and CWE-122 refer to buffer overflow on the stack and heap respectively.

```
void function(char *string){
    char buffer[24];
    // Unsafe - String might be longer than buffer
    strcpy(buffer, string);
}
```

Code Snippet 6: CWE-120 example

4.1.3 CWE-469

CWE-469 is a vulnerability that takes place when pointers are subtracted to determine size. If the pointers are not in the same memory chunk, the size will be invalid and might point outside of the allocated memory. An example of a program with this vulnerability can be seen in Code Snippet 7.

```
int size(int *start, int *end) {
    // Unsafe - End and start might not be in same memory chunk
    return end - start;
}
```

Code Snippet 7: CWE-469 example

4.1.4 CWE-476

CWE-476 is a NULL pointer dereference. In simpler terms, this occurs whenever a used pointer turns out to be NULL. An example of CWE-476 can be seen in Code Snippet 8.

```
int times(int *ptr, int num) {
    // Unsafe - Ptr might be NULL
    int i = *ptr;
    return i * num;
}
```

Code Snippet 8: CWE-476 example

4.1.5 CWE-Other

R. Russell et al. merely focused on classifying the top four CWE types. As such, they merged the remaining vulnerability types into one single class. Exactly which CWE types are present is unknown, however, CWE-20, CWE-457,

and CWE-805 are documented. It is very likely that some CWE types in this category are easier to detect than others.

4.2 Pre-processing

In order to create good representations of the source code, where only specific keywords are selected, it is necessary to parse the individual functions and group the different symbols and keywords. A custom pre-processing approach has been taken. To parse the functions, Clang is utilized. Clang is often only used to fully compile C/C++ programs, but in order to do so, it also parses the source code first. This parsing step using Clang's preprocessor does not require the code to be complete or functioning, and it can therefore be used on all functions in the VDISC dataset. One additional step is taken before the functions are sent to Clang for parsing. All statements starting with a hashtag (#), also called preprocessor directives, are processed and removed by the Clang. We do not want this as parts of code between *#if* statements may be removed due to missing environment variables. To counter this, a simple replacement occurs where all hashtags are replaced by "hash_", which means that "#if" becomes "hash_if". An example of a function and its output from the Clang pre-processor can be seen in Code Snippet 9. As one can see from this example, all keywords and symbols are successfully grouped and the value of the individual keywords are kept and ready to be filtered.

Approaches using abstract syntax trees and other tree-based methods, require the functions to be compilable to properly map types and function calls. Such approaches are therefore not possible in this case without discarding all functions that are not compilable. Using an approach that only looks at the function itself is therefore preferable with this kind of dataset. It is also more efficient, disregarding all outside context when pre-processing.

The Juliet Test Suite from Chapter 4.1 has also been used for experimenting. All vulnerable and benign functions were extracted before applying the same pre-processing as the VDISC dataset. All CWE types defined in Juliet that did not exist in VDISC were merged into the CWE-others category. It was then manually filtered to remove all functions with duplicate lexed representations.

4.3 Lexing and Vectorizing

After pre-processing, the functions are still lists of text, and therefore need to be processed further in order to become the input to a machine learning model. In this step, two methods of lexing are introduced, as well as a general method of vectorizing the functions into numbers instead of words.

The first lexed representation is of similar traits to the one described by R. Russell et al. This representation uses the output from Clang in order to keep all native C/C++ keywords but only include some identifier names. Russell et al. do not disclose what function, type, and variable names are included in their paper, but do mention that they included vulnerable functions from C standard libraries. The C standard libraries do not only contain functions that interact

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!\n";
}

/* Output from custom Clang pre-processor:

identifier 'hash_include'
less '<'
identifier 'iostream'
greater '>'
int 'int'
identifier 'main'
l_paren '('
r_paren ')'
l_brace '{'
identifier 'std'
coloncolon '::'
identifier 'cout'
lessless '<<'
string_literal '"Hello, world!\n"'
semi ';'
r_brace '}'
eof ''

*/
```

Code Snippet 9: Custom Clang pre-processor input and output

closely with memory in C, but also functions used in C++ code. As such, this lexed representation includes all C/C++ symbols and keywords, as well as all types and functions from all C standard libraries. Another handful of custom tokens has also been added, resulting in a total token count of 268, which is more than the 156 tokens used by Russel. In this lexing process, comments are removed, integers are split into separate digits and other types like floats and strings are mapped to placeholders. The total list of included tokens can be found in Appendix 1.

The second lexed representation is not taken from any of the previously reviewed literature. This method uses the raw function sources. The Clang representation removes all comments, spaces, and newlines. A different representation that keeps these features might be beneficial. The problem is that we cannot easily lex the code, as we do not know where to split the keywords and identifiers without a proper parser like Clang. A basic approach is taken where all brackets, braces, and operators get a space added on both sides before split-

ting the function source on all spaces. This retains most of the original data. The number of unique tokens in the dataset is still too many at this point for a practical embedding and is therefore reduced to tokens appearing in at least 1000 samples, which resulted in a total of 4253 unique tokens. These tokens include many of the same as the first lexed representation, however, they also include combinations due to lack of spaces, as well as library-specific types and functions. This representation is able to represent the code at a given token threshold without manually creating a list of whitelisted keywords.

Before these representations can be used in a neural network or another model, the lexed functions must be converted to integers. This is done by making a numbered vocabulary list from the 268 or 4253 tokens and then replacing the tokens with their index in the vocab list. A special padding token is also added to the vocabulary at index 0. This token is then added as padding to the end of each function to ensure a fixed length of 500. The padding is only performed where it is needed, and as such is not included in n-gram approaches.

The first and second lexed representation will from now be referred to as lexed-1 and lexed-2.

4.4 Machine Learning Approaches

In this chapter, several machine learning models trained on the lexed data are introduced. Details about each model will be explained, as well as details about what ideas did not work well, and were discarded under development. While exploring and experimenting the PR-AUC score was used to measure model performance.

4.4.1 Trivial Features using Random Forest

In the literature, there were some experiments done using simple statistical or trivial features, for example by Chernis et al. [16]. These features contain very little information about the actual contents of the source code they are extracted from, but they still reveal some information about things like the size, diversity, and complexity of the code. Additionally, statistical features are simple to understand and fast to extract from the source code. Therefore some experimenting with a small set of statistical features was done, even though the results in the literature were not very promising. A simple python script was used to extract the statistical features from the raw source code, and as such, no lexed representation is used. These features included character count, symbol count, character entropy, symbol entropy, character diversity, symbol diversity, and max nesting depth. In the aforementioned features, symbol refers to only non-alphanumeric characters.

4.4.2 BoW and N-grams using Random Forest

As was mentioned in section 2, n-grams and BoW-models are commonly applied when working with natural languages. One of the goals of this thesis was

to explore the effectiveness of applying these techniques to detect and classify vulnerabilities in a large dataset of program code. Both Chernis et al. [16], and Pang et al. [21] experiment with source code n-gram models, achieving somewhat promising results, although differences in datasets make it difficult to say anything conclusive about n-gram features for vulnerability detection. Despite program code largely consisting of the same characters as natural language text, programming languages are not natural languages and differ greatly from them in many ways. For example, in programming languages, identifiers like function and variable names can be anything, which means that functionally identical pieces of code, are not necessarily the exact same textually. This can cause issues when attempting to find specific patterns in code text that can help with identifying vulnerabilities.

If n-grams are extracted from the raw code text, the resulting vectorization becomes very large because identifier names are mostly unique from one function to another, resulting in a very large vocabulary. All these features are mostly useless since the specific n-grams containing the unique identifier names will only appear in one of the samples. To combat this, it is possible to use a minimum document frequency threshold and remove the n-grams that appear infrequently. The document frequency is defined as the fraction of documents that a specific term or n-gram appears in. However, a threshold can result in the opposite problem, as most n-grams containing identifiers are going to be removed, resulting in very few n-grams. This is a problem, as possibly useful code patterns are left out because the identifiers were different between the various samples. To overcome the aforementioned obstacles, the n-grams are extracted after the raw code text has been lexed. This solves the issues by removing most identifiers and replacing them with the same generic placeholder, only keeping specific identifiers from certain libraries that were believed to be useful.

The raw data from the VDISC-dataset was first lexed using the lexed-1 approach described previously. After lexing the raw code text, the n-grams were extracted from the lexed code texts. For $n > 1$, a minimum document frequency threshold was used to limit the total number of n-grams, as extracting every single n-gram that appeared in the data would simply be infeasible. At the same time, the goal was to extract as many as feasible, in order to have as many features as possible when performing feature selection. The idea is that some of the less frequent n-grams could be correlated with vulnerabilities, while the very infrequent n-grams simply occurred too rarely in the data to be useful. Various different configurations and combinations of n-grams from lexed data were experimented with. After extracting the n-grams, a chi-square feature selection was performed on the sets that exceeded 1024 n-grams, in order to reduce the feature space before classification.

4.4.3 Convolutional Neural Network

This approach using convolutional neural networks is inspired by the article by R. Russell et al. [9]. The approach is based upon sentence classification as

described in Chapter 2.6.2. Ze Zhang et al. have done extensive testing on what parameters should be considered for sentence classification problems [26], and as such the values that were chosen in this implementation are based on these recommendations in addition to experimenting. Both lexed-1 and lexed-2 are used to create two distinct CNN models.

To start, each unique token in the lexed dataset is embedded into a unique vector of dimensions either 13 or 64 depending on performance. The vectors are randomly initiated between -1 and 1 as Russell et al. found that pre-trained values using Word2Vec had little to no effect on the end results. Multiple convolutional layers and the combinations of these were tested. A single convolutional layer consisting of 512 filters, each with a filter length of 5, making 512 feature maps, followed by ReLU, proved to perform the best. The 512 feature maps each apply max-pooling along the total length of 500 (length of the input function with padding) before flattening the feature maps into 512 total features. A dropout of 0.5 is applied before a single linear layer reducing the output to the number of classes, which is 5 in the VDISC dataset. The optimizer used is ADAM with a learning rate of 0.001. This learning rate is a bit high, however, lower learning rates yielded similar results. The loss function used was binary cross-entropy with sigmoid applied to the input. Sigmoid is used over softmax as this is a multi-label problem where multiple CWEs can occur in a single function. One can also say that the network is trained on a shared feature representation. A summary of the network can be seen in Table 3.

Layer name	Output shape	Param count	Info
Embedding	[500, 13]	3,549	dim: 13 or 64
Conv1d (ReLU)	[512, 500]	33,792	kernel: 5
MaxPool1d	[512, 1]	-	kernel: 500
Flatten	[512]	-	-
Dropout	[512]	-	rate: 0.5
Linear	[5]	2,565	-

Table 3: Summary of CNN model

Russell et al. stated that class-weighing the vulnerable functions in the loss function was one of the keys to their performance. Different weights were explored, but a weight of around 10 : 1 for all vulnerable classes proved most efficient. Experiments revealed that multiple fully-connected layers significantly improved the CNN's predictions, however, the feature maps from the convolutions suffered in quality. The model trained the linear layers rather than optimizing the feature maps. In a scenario where one uses the CNN as a predictor, this works well, but according to Russel, random forest has proved to be better at classification. The linear layer is therefore just a single layer in order to prioritize training feature maps for deep representation learning. The classification results using this CNN are not optimized for direct vulnerability prediction, but rather deep representation learning. This is also reflected in Table 3, where the linear layer has a mere 2565 trainable parameters.

Both the CNN for lexed-1 and lexed-2 were trained for a total of 20 epochs. The best performing epoch for each of the CNNs were chosen as the final two models. Because this CNN approach produces two different networks based on the lexed input, the networks will from now on be referred to as CNN-1 and CNN-2, trained on lexed-1 and lexed-2 respectively.

4.4.4 Deep Representation Learning using Random Forest

As described in Chapter 2.6.2, neural nets can be used for representation learning, or in this case, deep representation learning. Much like the findings from R. Russell et al. [9], the CNN itself did not perform particularly well at classification. Instead, deep representation learning is used, and the 512 raw weights are extracted from the CNN before the dropout layer. These features were extracted from both CNN-1 and CNN-2 and were trained on using random forest.

From this point onward, the random forest models trained on CNN-1 and CNN-2 will be referred to as RF-CNN-1 and RF-CNN-2 respectively. A third model was also created. This model combines the 512 features from both CNN-1 and CNN-2 in order to create a random forest model with 1024 features. This model will be referred to as RF-CNN-3.

4.4.5 Other Combinations

Including the different methods described previously, other combinations were also tested. Among these are RF-CNN-3 combined with trivial features and different n-grams. None of these performed any better in combination. The feature overlap is likely too large to improve performance. With more than 1000 estimators using RF, the trivial features and n-grams actually decreased the final model performance. In addition to other combinations, the Juliet dataset was also merged with VDISC in order to create a larger set to train on. The Juliet dataset turned out to negatively affect the VDISC dataset. This is likely due to the test suite being artificial, as well as the VDISC labels being slightly mislabeled due to the static analysis performed to create the labels. These combinations, as well as the Juliet dataset, will not be discussed further as they did not positively affect performance and further research was dropped.

4.5 Experimental Setup

4.5.1 Resources

All machine learning models and pre-processing was run on resources provided by NTNU IDI. The server used had the following specifications:

CPU	2x Intel Xeon Gold 6132
GPU	2x Nvidia Tesla V100 32 GB
RAM	768 GB
CUDA	Version 11.6

All the different resources are used to their full extent at different stages of the approaches. The pre-processing utilizes significant amounts of memory. The CNN utilizes almost 100% of both GPUs, and RF utilizes close to 100% of all CPU resources. The CPU and GPU resources are not required to reproduce the results, however, the memory is important to reproduce the best results using RF.

4.5.2 Libraries and Software

Different Python libraries and software were used to implement the pre-processing as well as the models. All the software and libraries that were used are open-source projects. For creating the neural networks, PyTorch was used. PyTorch is a high-performance deep learning library for Python with great possibilities for GPU acceleration [27]. For NLP-based processing, PyTorch-NLP was used [28]. This library includes text encoders that are used extensively to convert vectors of words into vectors of numbers. Scikit-learn is a library that provides machine learning tools and models, and was used along with matplotlib for evaluation and for the random forest implementation [29] [30]. To make handling matrices easier, the libraries Pandas and NumPy were used [31] [32].

5 Results

In this chapter, the results of the different methodologies will be presented. The results include relevant metrics and hyper-parameters for each approach. Because both vulnerability detection and CWE classification are important, all results include both a multi-label classification of CWEs and a binary classification (except CNN-1 and CNN-2). The binary classification is done by merging all CWE types into one "vulnerable" y-value. The binary classification is only performed on methods using random forest. As mentioned in the theory part, multi-label and multi-class are different. Multi-label models can predict multiple vulnerability types at the same time, while multi-class can have one of many types. The CNN models are strictly trained on multi-label data. All metrics that require a selected threshold, like F1 and MCC, have their threshold calculated on the validation set. All methods utilizing random forest have used 100 estimators and entropy as the gain function unless another configuration is specified. The tables containing the different evaluation metrics will contain the two metrics called *ROC W* and *PR W*. *W* in this case refers to the metric being weighted. This means that the score is calculated based on the number of vulnerable samples in each class, rather than an average over the number of classes. Note that both ROC and PR refer to the AUC score.

5.1 Trivial Features using Random Forest

The small set of trivial statistical features was primarily created to be used in conjunction with other feature sets, however, the trivial feature set alone was also used to train two different random forest models, one for multi-label and one for binary classification referred to as RF-trivial and RF-Trivial-Binary respectively. Each model consisted of 100 estimators. The results of both models can be seen in Table 4. RF-Trivial-Binary achieved twice as high PR-AUC as RF-Trivial.

Model	ROC	ROC W	PR	PR W	MCC	F1	ACC
RF-Trivial(100)	0.679	0.692	0.072	0.082	0.108	0.122	95.42
RF-Trivial-Binary(100)	0.680	-	0.146	-	0.143	0.208	83.66

Table 4: Metrics for RF multi-label and binary models trained using trivial features

5.2 BoW and N-gram using Random Forest

A total of 5 different n-gram feature sets were extracted from the lexed dataset. A minimum document frequency of *0.001* was used for all feature sets, except for 1-grams, as there were only 268 possible tokens. Table 5 shows the different n-gram feature sets that were tested, including the number of n-grams that each yielded. In order to reduce the number of features before training, the n-gram

features of each feature set were ranked with the chi-squared-test, and only the top 1024 features for each set were kept.

n	n-grams
1	268
2	1328
3	3872
4	7376
1-4	12757

Table 5: N-gram combinations and number of extracted n-grams

To test the various feature sets, random forest models were used. For each of the models, the Scikit-learn RandomForestClassifier class was used with 100 estimators and entropy as the splitting criterion. The various models for the feature sets in table 5 will be referred to as RF-N1, RF-N2, RF-N3, RF-N4, and RF-N1_4 respectively. The results of all models are shown in Table 6. The best performing model was RF-N3. A second model based on the best performing model was also created, with the same parameters as RF-N3, except with 1000 estimators, instead of 100, and is included at the bottom of Table 6.

Model	ROC	ROC W	PR	PR W	MCC	F1	ACC
RF-N1(100)	0.891	0.904	0.342	0.394	0.409	0.413	97.62
RF-N2(100)	0.898	0.911	0.378	0.432	0.446	0.449	97.77
RF-N3(100)	0.902	0.914	0.390	0.443	0.463	0.464	**97.90**
RF-N4(100)	0.897	0.911	0.386	0.438	0.455	0.457	97.77
RF-N1_4(100)	0.900	0.911	0.383	0.436	0.459	0.457	97.76
RF-N3(1000)	**0.930**	**0.930**	**0.415**	**0.463**	**0.471**	**0.473**	97.88

Table 6: Metrics for all RF multi-label models trained using n-grams

The same feature sets were also used to train models for binary classification on combined class labels. These models have the binary suffix and the results can be seen in Table 7.

Model	ROC	PR	MCC	F1	ACC
RF-N1-Binary(100)	0.884	0.469	0.477	0.510	92.56
RF-N2-Binary(100)	0.895	0.499	0.505	0.537	**93.30**
RF-N3-Binary(100)	0.899	0.506	0.513	0.543	92.98
RF-N4-Binary(100)	0.895	0.500	0.506	0.537	93.00
RF-N1_4-Binary(100)	0.893	0.496	0.507	0.537	92.95
RF-N3-Binary(1000)	**0.907**	**0.522**	**0.516**	**0.547**	93.22

Table 7: Metrics for all RF binary models trained using n-grams

In both multi-label and binary classification, the 3-gram feature set yielded the best performing models. Figure 3 provides a visualization of the 100 most

important n-grams, as scored by the chi-squared test. A larger font corresponds to a higher score.

Figure 3: Visualization of the 100 most important 3-grams scored by the chi-squared-test

5.3 Convolutional Neural Network

Classifying performance for the CNN alone was not prioritized in this thesis, as the real goal was deep representation learning. However, optimizing the models was still a priority. Only multi-label models classifying CWE types were trained. The only varying parameter for the CNNs was the embedded dimension. Dimensions of 13 and 64 were applied to both lexed-1 and lexed-2, in order to decide which dimension fit better for which representation. The evaluation metrics for CNN-1 and CNN-2 for both dimensions can be seen in Table 8. For CNN-1 (lexed-1), an embedded dimension of 13 performed better, while 64 dimensions performed better for CNN-2 (lexed-2). This comparison is done by looking at PR-AUC values only.

Model	ROC	ROC W	PR	PR W	MCC	F1	ACC
CNN-1(13)	0.919	0.920	0.318	0.375	0.427	0.425	97.59
CNN-1(64)	0.916	0.918	0.316	0.375	0.422	0.419	97.56
CNN-2(13)	0.921	0.926	0.349	0.404	0.443	0.441	97.63
CNN-2(64)	0.897	0.916	0.354	0.413	0.439	0.440	97.70

Table 8: Metrics for CNN-1 and CNN-2 with embedding dimensions 13 and 64

Both networks were trained for a total of 20 epochs, and the best epochs among these were selected. For CNN-1 and CNN-2 this corresponds to epochs 18 and 20 respectively. After these epochs, the model loss increases in both cases,

and it started to overfit. Lower learning rates helped counter the overfitting, however, the model PR performance did not improve.

5.4 Deep Representation Learning using Random Forest

The three different deep representation learning models RF-CNN-1, RF-CNN-2, and RF-CNN-3 were all trained with RF using 100 estimators as stated earlier. Two different methods of measuring the quality of a random forest split were tried: Gini and entropy. Entropy proved to consistently outperform gini and was therefore used. Using class weights proved only to reduce performance over manually selecting thresholds after the fact. The best model RF-CNN-3 reached PR-AUC scores of 0.449 and 0.555 on the multi-labels and binary labels respectively. These are the best scores achieved out of all the models trained with 100 estimators. In an attempt to further increase these scores, the estimator values were expanded to the values $[100, 1000, 2500]$ for RF-CNN-3 and RF-CNN-3-Binary. The results for the multi-label models and binary models can be seen in Table 9 and Table 10 respectively.

Model	ROC	ROC W	PR	PR W	MCC	F1	ACC
RF-CNN-1(100)	0.910	0.920	0.434	0.484	0.499	0.500	98.01
RF-CNN-2(100)	0.907	0.918	0.424	0.481	0.495	0.496	97.90
RF-CNN-3(100)	0.913	0.925	0.449	0.505	0.514	0.514	98.00
RF-CNN-3(1000)	0.939	0.940	0.468	0.519	0.522	0.522	98.01
RF-CNN-3(2500)	**0.942**	**0.942**	**0.468**	**0.520**	**0.523**	**0.524**	**98.02**

Table 9: Metrics for all RF-CNN multi-label models

Model	ROC	PR	MCC	F1	ACC
RF-CNN-1-Binary(100)	0.906	0.538	0.548	0.577	**93.78**
RF-CNN-2-Binary(100)	0.906	0.534	0.544	0.572	93.63
RF-CNN-3-Binary(100)	0.915	0.555	0.564	0.589	93.69
RF-CNN-3-Binary(1000)	0.920	0.567	**0.568**	**0.593**	93.72
RF-CNN-3-Binary(2500)	**0.921**	**0.568**	0.567	0.591	93.68

Table 10: Metrics for all RF-CNN Binary models

The improvement of increasing the random forest estimators can be seen in Figure 4. This figure showcases a PR curve for the lower 100 estimators model versus the higher 2500 estimators model. The PR score improved significantly from 100 to 2500 estimators for both the multi-label and binary models. All the metrics calculated for both models using 2500 estimators can be found in Appendix 2.

Considering the fact that CNN-RF-3 (multi-label and binary) is the best performing model, it is natural to include the confusion matrices for each CWE type. As for the multi-label classifier, Table 11, 12, 13, 14, 15 presents the

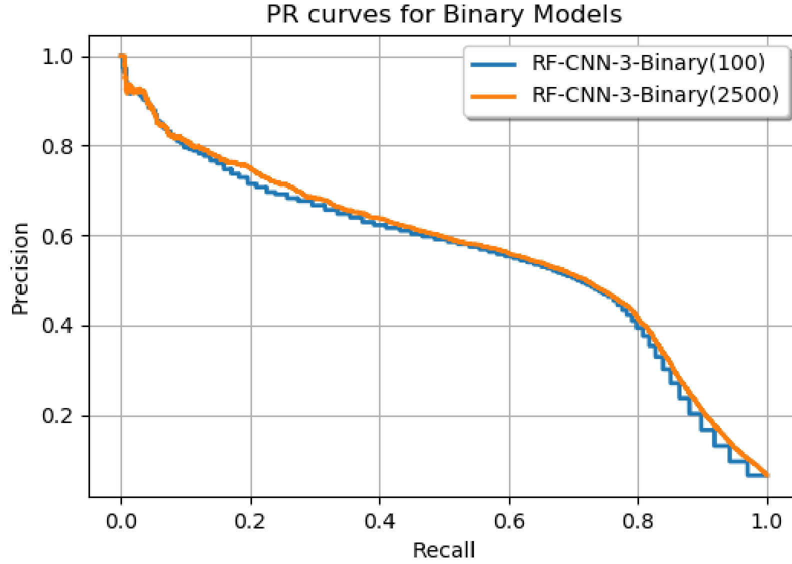


Figure 4: PR curves for RF-CNN-3 at 100 and 2500 estimators

confusion matrices for the types CWE-119, CWE-120, CWE-469, CWE-476, and CWE-other respectively. Because this is a multi-label model, one can not simply create a single confusion matrix. The reason for this is once again that multiple labels can be true at the same time. A such combined confusion matrix would not make sense. The confusion matrix for the binary classification, or detection, can be seen in Table 16.

		Predicted Label	
		Negative	Positive
True Label	Negative	122903	2064
	Positive	529	1923

Table 11: CWE-119 Confusion Matrix for best model (multi-label)

		Predicted Label	
		Negative	Positive
True Label	Negative	118434	4094
	Positive	1056	3835

Table 12: CWE-120 Confusion Matrix for best model (multi-label)

		Predicted Label	
		Negative	Positive
True Label	Negative	126925	216
	Positive	181	97

Table 13: CWE-469 Confusion Matrix for best model (multi-label)

		Predicted Label	
		Negative	Positive
True Label	Negative	125915	312
	Positive	584	608

Table 14: CWE-476 Confusion Matrix for best model (multi-label)

		Predicted Label	
		Negative	Positive
True Label	Negative	121948	1981
	Positive	1573	1917

Table 15: CWE-other Confusion Matrix for best model (multi-label)

		Predicted Label	
		Negative	Positive
True Label	Negative	113552	5614
	Positive	2434	5819

Table 16: Confusion Matrix for best model (binary)

5.5 Result Summary

In order to put into perspective how each of the proposed methodologies performed, it is important to compare their PR curves as well as ROC curves. In Figure 5, the best performing models from each class (using 100 RF estimators) have their PR curve displayed. RF-Trivial is the only outlier performing significantly worse than the rest. In Figure 6, the same models have their ROC curves displayed. Once again RF-Trivial is the only model to severely underperform.

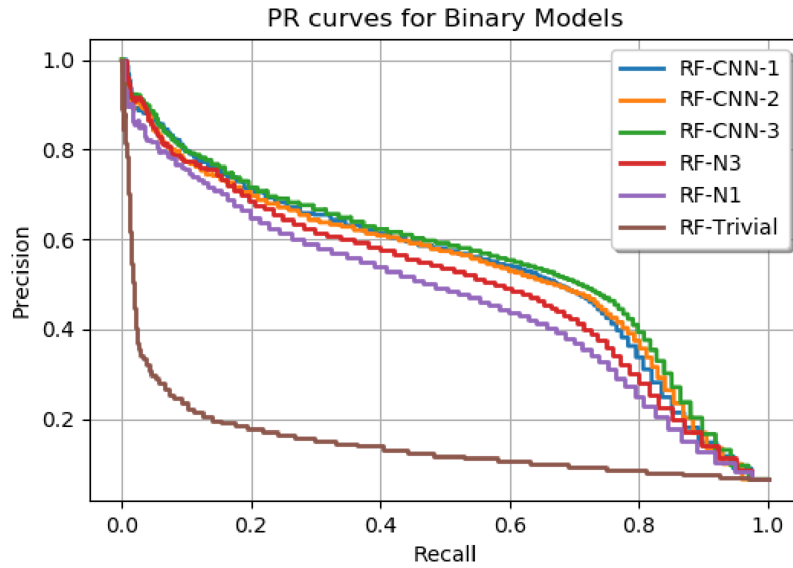


Figure 5: PR curves for selected binary RF models (100 estimators)

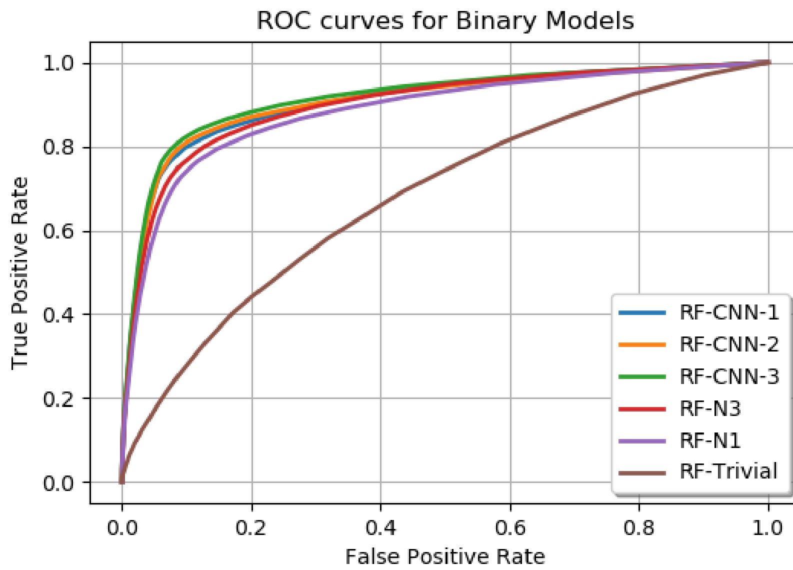


Figure 6: ROC curve for selected binary RF models (100 estimators)

6 Discussion

This chapter covers a discussion of the results in the previous chapter. The models will be discussed, compared to literature and outstanding challenges will be addressed.

6.1 Models

The trivial statistical features performed by far the worst out of all the models tested, achieving a PR-AUC of only 0.072 for the multi-label classification. This was not very surprising, as most of these features say very little about the underlying code and this feature set was by a large margin the smallest set tested. On the binary classification, it performed better, although still subpar, with a PR-AUC of 0.146. A larger set of trivial statistical features could potentially yield better results up to a certain point, although it is unlikely that trivial statistical features could outperform more complex features. Program code is simply too syntactically intricate. It is also important to remember that the difference between a non-vulnerable and vulnerable code sample can be very small, and good features must be able to capture this. Trivial features similar to the ones tried in this thesis were used in Chernis et al. [16], achieving similar results, with trivial features performing quite a bit better than randomly guessing, but still not very impressive compared to the other models in this thesis.

Bag-of-words and n-grams encompasses larger and more complex statistical features and generally performed quite well in the literature. Just using the frequencies of each individual token (1-grams) yielded a PR-AUC of 0.342, which is a massive improvement over the trivial statistical features. Extending to 2-grams gave a good improvement, reaching a PR-AUC of 0.378, and 3-grams performed better still, yielding the highest scores of the tested n-gram models with a PR-AUC of 0.390 using 100 estimators. 4-grams performed only slightly worse. Quite surprisingly, selecting the top n-grams from the combined pool of 1-4-grams also performed slightly worse than just the top 3-grams. This suggests that perhaps the chi-square test did not actually find all the "best" features. One reason for this could be that there was some overlap between differently sized n-grams. Although it should be noted that the difference in performance was very small. When increasing the number of estimators to 1000, the 3-gram model saw a further increase in PR-AUC to 0.415.

Relating to **RQ3**, Figure 3 displays some of the best 3-grams, and by extension also which tokens were important when detecting vulnerabilities. The most prominent 3-gram is *char identifier l_square* which simply indicates the declaration of an array of characters. In C/C++, arrays are stored as contiguous memory blocks, and several common vulnerabilities relate to array operations and improper restrictions of their boundaries. As such, it makes sense that there is a correlation between functions dealing with arrays and functions containing vulnerabilities. Similarly, *buffer* is a custom token, resulting from a few different identifier names in the code related to the word "buffer", which also indicates

operations related to arrays. A number of tokens from library functions related to memory operations and file operations, for example, *memcpy*, *strcmp*, *sizeof*, and *fopen*, are also prominent in some of the top 3-grams. It is difficult to draw accurate conclusions about why these specific token combinations are important, but certain tokens do give some information about the underlying purpose of the code.

The convolutional neural network models; CNN-1, CNN-2, and CNN-3 do not perform on par with n-grams when it comes to classification performance. Looking back on Table 8, the PR-AUC was recorded at 0.318 and 0.354 for CNN-1 and CNN-2 respectively, while the 3-grams got a value of 0.390 using only 100 estimators. The mediocre performance was expected as the linear classification layers of the model have been limited to a single layer. CNN-2 produces significantly better results than CNN-1 in our results. This is likely because the larger lexed-2 vocabulary contains tokens that are closely related to vulnerabilities, and the simple linear layer easily manages to capture this. CNN-1 is trained on lexed-1 with a smaller vocabulary, and will therefore need more neurons to better understand the real semantics of the code. After all, these models are trained using as few linear layers as possible in order to force the other weights to update and produce good features for representation learning. As for the embedded dimensions for both models, the results do not come as a surprise. When the vocabulary size increases, the model also benefits from an increase in the embedded dimensions. A too-large dimension will just require more training. Only a multi-label implementation was considered for the CNNs because of time restrictions and complexities.

The deep representation learning approaches perform the best out of all models implemented in this thesis. As expected, the CNN-1 features perform very similarly to the CNN-2 features when allowed to properly train its classifier, random forest in this case. The large vocabulary behind RF-CNN-2 works better for detecting vulnerabilities deeply connected to specific tokens, but RF-CNN-1 retains more semantic information due to the generalized vocabulary. Combining these two makes RF able to learn from both representations, and perform even better. This hybrid approach resulted in our best model for multi-label as well as binary classification, RF-CNN-3, trained with 2500 estimators. This model achieved a PR-AUC of 0.468 and 0.568 on the multi-label and binary results respectively. Compared to the best n-grams model, our second-best approach, this model achieves an increase in PR-AUC of 5.3% on multi-class and 4.6% on binary classification. Looking at Table 16 again, we can extract a detection rate along with a relative rate of false positives for the binary model. 70.5% of all vulnerable functions in the test set are detected, and for every detected function 0.96 false positives are generated. Assuming that the distribution of vulnerable samples in the test set is representative of real-world code, this is a very acceptable result. The multi-label model achieved worse results, yet good compared to our other models, and two reasons can explain this. First of all, the multi-label model needs to learn separate features for each class, also called a shared feature representation. Using the same models for both multi-label and binary classification clearly limits the multi-label case. The second

explanation is that the multi-label models predict the wrong class, but does indeed predict that the function is vulnerable. Take CWE-119 and CWE-120 for instance, these two classes can have fairly similar code semantics. When classifying binary, the classes are not considered, and this case will simply be labeled vulnerable, thus resulting in an increase in the evaluation metrics.

Generally, for random forest, a higher number of estimators yielded better results. Random forest is great in this way, as it does not overfit. However, the increase in estimators does come at a cost. The computation time scales with the estimators, and so does the model size. Anything above 2500 estimators would take up several gigabytes of memory. The increase in estimators from 1000 to 2500 for RF-CNN-3 did not prove a significant increase in performance compared to moving from 100 to 1000, indicating that the model is plateauing.

Looking back at the multi-label confusion matrices in Table 11, 12, 13, 14, 15, we can see that the difficulty of detecting some CWE types is harder than others. This is caused by the variance in different vulnerabilities, sometimes occurring on a single line and other times consisting of complex semantics. CWE-119 and CWE-120 perform very similarly, but this makes sense as both CWE types have similar semantics. The vulnerability type our model has the hardest time detecting is CWE-469. This vulnerability is simple for a human to detect, but harder for a machine learning model to understand, as semantics are very important. The model needs to understand that two variables in a subtraction are pointers. Such an understanding is hard to achieve but could be done better by using value flow graphs or other graph-based representations like described by Cheng et al. [20]. This would, however, require a dataset parseable by the respective graph extractor tools.

Having implemented, evaluated, and discussed machine learning approaches to vulnerability classification using statistical features, n-grams, and token-based representation learning, **RQ1** has already been answered. Looking back at Figure 5 and 6 showing PR and ROC respectively, we can see the differences between the distinct approaches. Statistical features do not perform well by themselves, but do give some insight and are better than a random guess. N-grams perform surprisingly well, just a small step down from using deep representation learning. Unsurprisingly, BoW performs worse than n-grams, as BoW capture no semantics. Out of the n-grams, 3-grams perform the best for our dataset.

In addition to evaluating the different approaches using statistical features, n-grams, and token-based representation learning, hybrid combinations have also been explored. Combining statistical features and n-grams with our best non-hybrid model, RF-CNN-1, proves that the overlap in understanding is about 100%, rendering a hybrid approach redundant. A hybrid approach using two lexed representations and deep representation learning for each of them proved itself excellent. This approach reached the highest results in all of our experiments and beat all pure NLP approaches in the referenced literature. The answer to **RQ2** is that hybrid approaches do perform better, however, both methods need to capture enough semantic information, as well as have minimal overlap.

6.2 Dataset

The NLP-only approaches in this thesis were taken both to keep complexity and processing times down, but also to be able to use any dataset. The VDISC dataset used in this thesis only contains functions without any outside code context. This severely restricts the methodologies that can be used to more statistical and NLP-based approaches. For our approach, the dataset has performed well, but the reviewed literature has proven that graph and tree-based approaches also perform well, and likely learn different and more semantic features [20] [13].

One positive of the VDISC dataset is its sheer size. It contains over a million function samples. The labels however are made by static analysis. This does not mean much for our models, as they can always be retrained on new data. Our methodologies prove that our models can successfully extract context from the functions. If significant changes are made to the tokens in C/C++, the dataset will need updating. One problem using this dataset might be the bias toward specific code bases. The VDISC dataset consists of 35% code from Debian. Tokens specific to Debian are especially prominent in the lexed-2 representation. This means that real-world performance outside Debian source code will likely perform worse using the lexed-2 representation. Our lexed-1 representation addresses this problem as it merely contains C/C++ keywords, standard library functions and a select few custom tokens.

One downside to using the pre-determined VDISC train-val-test split is that cross-validation is not used. Cross-validation is often used to tune the hyper-parameters, as well as a method of generalization to counter over-fitting. However, in the case of VDISC, the size of the dataset is relatively large. Whether it would benefit from cross-validation was not tested in order to produce results that would be comparable with the literature.

6.3 Comparison to Literature

As we know from Chapter 2.6.5, evaluation metrics can be skewed, especially on unbalanced datasets. In the case of the dataset used in this thesis, this is also true. Comparing literature using dramatically different datasets will not provide good comparisons. However, in the reviewed literature there are three different articles covering model implementations on top of the VDISC dataset. These implementations include the one by R. Russell et al. [9], which the deep representation learning approach in this thesis is based upon, Z. Bilgin et al. [12] and J. Harer et al. [10].

In the results section by R. Russell et al., they do not provide any multi-label metrics other than PR Curves showcasing the different CWE types. It is therefore hard to compare the multi-label performance. The PR Curves for RF-CNN-3(2500), the best performing model in this thesis, have in Figure 7 been overlayed over Russel's results. The graph is cluttered, but the vibrant colored graphs represent our model, while the stapled and non-vibrant colors represent theirs. The difference is quite staggering, where our model clearly

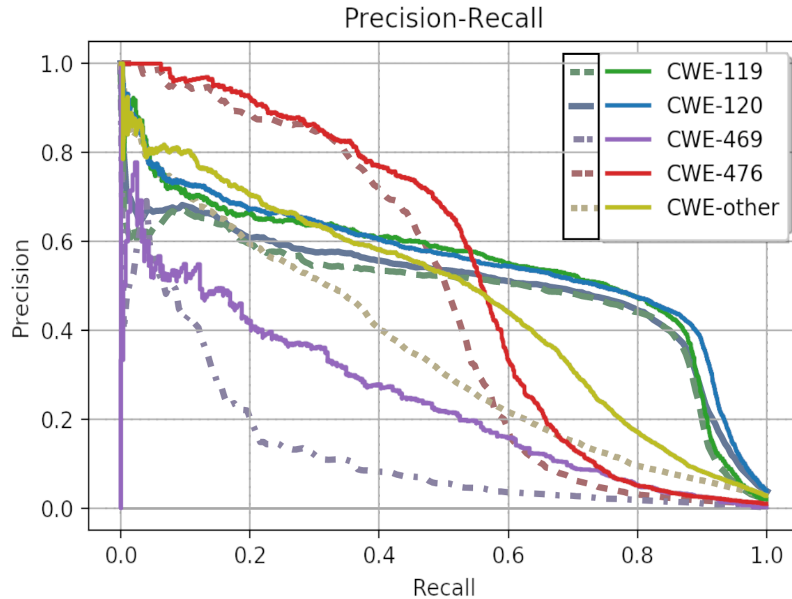


Figure 7: PR AUC for best multi-label model layered over results by R. Russell et al. [9]. All dotted lines, as well as the dull blue continuous line, represent the work by Russell et al. The vibrant continuous lines represent the results in this thesis.

outperforms on classes like CWE-469 and CWE-Other. The PR score is closer for CWE-119, CWE-120, and CWE-376, however, it still performs significantly better. Removing most of the linear layers at the end of the CNN likely had a big impact on these results. For CWE-Other the automatic lexer-2 provided extra context. Using only a fixed vocabulary has proved excellent, however, using automatically picked tokens likely helped classify the many different CWE categories in CWE-other. The lexed-1 representation also includes more tokens that the original proposition by Russel.

As for binary classification, Russell et al. provided several metrics. In Table 17, our best model is represented both for binary and multi-label classification along with results from the literature. Looking at the table the multi-label model, RF-CNN-3 outperforms the binary model by Russell in ROC and PR by a little. This is impressive, as this model is trained on a multi-label representation and not a binary one. Our binary model RF-CNN-3-Binary outperforms every model in the table on PR, MCC, and F1. Comparing the PR AUC, our model reaches a whole 5% above the baseline by Russel. Similar values are seen in MCC and F1 with about a 3% improvement. These values may not seem like too much, but on such a complex topic as vulnerability detection using only semantic features from functions, this is a good improvement.

Model	ROC	PR	MCC	F1
R. Russell et al. [9]	0.904	0.518	0.536	0.566
Z. Bilgin et al. [12]	0.804	-	-	0.414
J. Harer et al. [10]	0.870	0.490	-	-
RF-CNN-3(2500)	**0.942**	0.520	0.523	0.524
RF-CNN-3-Binary(2500)	0.921	**0.568**	**0.567**	**0.591**

Table 17: Our best models compared to literature. Weighted averages are used.

In Table 17, Bilgin et al. and Harer et al. are also represented. Bilgin et al. used a multi-label methodology, while Harer et al. focused on binary classification. Both of their results are worse than the results by R. Russel, which means their AST and CNN/ET implementations did not perform as well on the VDISC dataset. Based on the metrics that were available, it would seem that our best model performs better even when classifying multi-label data. As for the AST implementation by Bilgin et al., this makes sense. In their implementation, they severely under-sample the dataset in order to only include AST-parseable functions. Our multi-label RF-N1(100) model, essentially a BoW model, achieved an almost identical F1 score. This means that an under-sampling of the VDISC dataset for AST extraction is likely not a good approach to the problem.

6.4 Outstanding Challenges

There are several outstanding challenges to using machine learning for static vulnerability detection in source code. Based on the reviewed literature and our own experiments, two main challenges that could be considered to be the most important for this problem have been identified.

6.4.1 Lack of Data

The most important part of any supervised machine learning is the data used for actual learning. A good data set needs to be large enough and contain enough samples of all classes in order to sufficiently train the model. It needs to be diverse enough so that the model becomes generalized and is able to accurately classify data from different sources. In this thesis, the data consists of source code, which has both upsides and downsides. A positive is that there is a lot of source code available that can potentially be used as training and testing data. A potential challenge is the fact that the difference between a "good" piece of code and a code with a security vulnerability can be very tiny. This also means that security vulnerabilities can be hard to spot, which makes manually labeling data very time-consuming. Some of the reviewed literature used existing static analyzers in order to generate a labeled data set from publicly available source code. This approach has the advantage that it can create a data set from any piece of source code, however, it also means that only the vulnerabilities detected by the static analyzers will be found and labeled correctly. If the goal is to achieve better performance than the available static analyzers, this approach is

simply not an option because the data would be too noisy; containing instances of buggy code that is labeled as clean and vice versa. Taking the aforementioned reasons into consideration, we believe that obtaining a good data set is one of the major challenges in using machine learning for vulnerability detection. That being said, there are several publicly available data sets on the topic, some of them containing actual code, while others contain specific features extracted from code.

6.4.2 Feature Engineering

In order to create a useful machine learning model, we need to extract a good set of features from the data. As mentioned, the difference between vulnerable code and good code can be minuscule, so we need features of a fine enough granularity in order to capture these tiny differences. Additionally, security vulnerabilities in code can present in a range of different ways, which possibly means that we require a range of different types of features to accommodate them all. Some vulnerabilities may be captured by statistical or token-based features, while others require more complex features that capture the context of the code. This has been proved by the methodologies implemented in this thesis. As seen in Chapter 2.5, there are a lot of different features that can be extracted from code, and finding a good feature set that is able to be used for accurately classifying security vulnerabilities is one of the big challenges. From the methodologies restricted to natural language, deep representation learning on lexed tokens proved to perform well, however, the results are still far away from fully automating vulnerability detection. More feature engineering and better data are needed.

7 Conclusion and Further Work

7.1 Conclusion

With the rise of technology, computers now play an important role in our lives. Software systems are now responsible for performing essential tasks and handling personal data concerning billions of people. It is, therefore, more important than ever that such systems are secure and behave as expected. Detecting vulnerabilities during development play a significant part in securing these systems. At the start of this thesis, a goal was set to research how accurately and efficiently vulnerabilities in C/C++ functions could be detected and classified using machine learning. Through a literature review, reproduction of methodologies, and contributions of our own, the goal was successfully achieved. After reviewing the literature, a choice was made to narrow the scope of our methodologies to natural language-based methods. A large C/C++ dataset containing vulnerable and benign functions was chosen to fit this new scope. This dataset has been processed by two lexers, as well as a custom statistical analysis. Three distinct machine learning approaches were taken which include convolutional neural networks, deep representation learning, and random forest using n-grams and statistical features.

Based on previous successful methodologies from the literature, this thesis developed a system for detecting and classifying vulnerabilities in source code by using custom lexers and various NLP-based feature engineering methods. Using deep representation learning and a random forest classifier, this thesis provided a model that was able to outperform the reviewed literature on the same dataset. For binary classification, this model managed to achieve a detection rate of over 70% while producing fewer than one false positive for each real vulnerability found. A very simple method of using 3-gram features and a random forest classifier is also provided in this thesis. A model using this method also provided results comparable to the literature, while reducing computational complexity. Despite the performance uplifts provided in this thesis, there is still a great deal of room for improvements in general when it comes to detection and classifying vulnerable code using machine learning. To further see large improvements, a combination of more accurate data collecting and labeling, more complex feature sets, and more powerful models would probably be required. However, the aim of this thesis was never to achieve perfection, but rather to explore possible alternatives to heavy, traditional vulnerability detection methods.

7.2 Future Work

7.2.1 Better Data Collection

In chapter 6.4.1, the lack and importance of good data were addressed. Manually labeling data is too time-consuming, and code labeled by static analysis is not optimal as the number of undetected cases, false negatives, is often too high. While researching for this thesis, G. Bhandari et al. published their paper called *CVEfixes* [33]. A CVE is a recorded vulnerability in some published software. Their paper proposes a method for automatically collecting the vulnerable code and fixed code of all registered CVEs in the public U.S. National Vulnerability Database (NVD). Implementing this solution and using the vulnerable functions and fixes as data for our models is a great step in the right direction.

7.2.2 Incorporate Tree/Graph-based Methods

As stated at several points throughout this thesis, tree and graph-based models are widely used on the topic of vulnerable source code. Due to our selected dataset being scoped to individual functions only, this method is hard and in some cases impossible to implement. However, custom parsers can be developed, and new datasets can be generated. Combining graph or tree-based methods with our current models will provide interesting and likely better results.

7.2.3 Detect First Then Classify

A logical next step in attempting to achieve better performance on the vulnerability classification would be to first utilize binary classification to detect which samples contain vulnerabilities and which do not. Our current models primarily use multi-label classification to distinguish CWE types, while some do binary classification in addition. The binary results proved to have a higher detection rate. Therefore, by first detecting with binary classification, we can ensure a higher detection rate, and then sequentially achieve the same or better CWE classification. This also means that our CNN-1 and CNN-2 approaches could be converted to binary classification models.

References

[1] Scott Christiansen and Mayana Pereira. *Secure the software development lifecycle with machine learning.* https://www.microsoft.com/security/blog/2020/04/16/secure-software-development-lifecycle-machine-learning/, Accessed November 9, 2021. 2020.

[2] Microsoft. *Code analysis for C/C++ overview.* https://docs.microsoft.com/en-us/cpp/code-quality/code-analysis-for-c-cpp-overview?view=msvc-170, Accessed November 9, 2021. 2021.

[3] *Clang Static Analyzer.* https://clang-analyzer.llvm.org/, Accessed November 9, 2021. 2021.

[4] Jonathan Corbet. *An update on the UMN affair.* https://lwn.net/Articles/854645/, Accessed November 9, 2021. 2021.

[5] *The Origin of ANSI C and ISO C.* https://blog.ansi.org/2017/09/origin-ansi-c-iso-c/, Accessed May 18, 2022. 2017.

[6] *Most Popular Technologies.* https://insights.stackoverflow.com/survey/2020, Accessed May 18, 2022. 2021.

[7] *Common Weakness Enumeration.* https://cwe.mitre.org/index.html, Accessed November 9, 2021. 2021.

[8] Yoav Goldberg and Omer Levy. "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method". In: *arXiv preprint arXiv:1402.3722* (2014).

[9] R. Russell et al. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE international conference on machine learning and applications* (2018).

[10] J. A. Harer et al. *Automated software vulnerability detection with machine learning.* arXiv preprint arXiv:1803.04497. 2018.

[11] Xin Li et al. *Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning.* Applied Sciences 10.5 (2020): 1692. 2020.

[12] Z. Bilgin et al. *Vulnerability prediction from source code using machine learning.* IEEE Access, 8, 150672-150684. 2020.

[13] Z. Li et al. *A comparative study of deep learning-based vulnerability detection system.* IEEE Access, 7, 103184-103197. 2019.

[14] Yoon Kim. *Convolutional Neural Networks for Sentence Classification.* 2014. DOI: 10.48550/ARXIV.1408.5882. URL: https://arxiv.org/abs/1408.5882.

[15] Davide Chicco and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation". In: *BMC genomics* 21.1 (2020), pp. 1–13.

[16] B. Chernis and R. Verma. “Machine learning methods for software vulnerability detection”. In: *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics (pp. 31-39)* (2018).

[17] K. Heo, H. Oh, and K. Yi. “Machine-learning-guided selectively unsound static analysis”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 519-529)* (2017).

[18] T. Chappelly et al. “Machine Learning for Finding Bugs: An Initial Report”. In: *2017 IEEE workshop on machine learning techniques for software quality evaluation (MaLTeSQuE) (pp. 21-26)* (2017).

[19] Joern. *Joern - The Bug Hunter’s Workbench*. `https://joern.io/`, Accessed November 9, 2021. 2021.

[20] X. Cheng et al. “DeepWukong: Statically detecting software vulnerabilities using deep graph neural network”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM), 30(3), 1-33* (2021).

[21] Y. Pang, X. Xue, and H. Wang. *Predicting vulnerable software components through deep neural network*. In Proceedings of the 2017 International Conference on Deep Learning Technologies. 2017.

[22] *Cppcheck - A tool for static C/C++ code analysis*. `https://cppcheck.sourceforge.io/`, Accessed November 9, 2021. 2021.

[23] *Flawfinder*. `https://dwheeler.com/flawfinder/`, Accessed November 9, 2021. 2021.

[24] samate@nist.gov. *Juliet Test Suite for C/C++*. `https://samate.nist.gov/SRD/testsuite.php`, Accessed May 18, 2022. 2017.

[25] CWE Content Team. *CWE VIEW: Weaknesses in Software Written in C*. `https://cwe.mitre.org/data/definitions/658.html`, Accessed May 18, 2022. 2008.

[26] Ye Zhang and Byron Wallace. “A sensitivity analysis of (and practitioners’ guide to) convolutional neural networks for sentence classification”. In: *arXiv preprint arXiv:1510.03820* (2015).

[27] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[28] Michael Petrochuk. *PyTorch-NLP: Rapid Prototyping with PyTorch Natural Language Processing (NLP) Tools*. `https://github.com/PetrochukM/PyTorch-NLP`. 2018.

[29] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[30] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[31] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`.

[32] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: `10.5281/zenodo.3509134`. URL: `https://doi.org/10.5281/zenodo.3509134`.

[33] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software". en. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*. ACM, 2021, p. 10. ISBN: 978-1-4503-8680-7. DOI: `10.1145/3475960.3475985`.

Appendices

1. List of Manually Selected Tokens

This is a list of tokens that were used to lex the functions for lexed representation 1. Numbers (0-9) are not included. Tokens in parenthesis are merged into the single token before the parentheses.

```
comment identifier raw_identifier numeric_constant char_constant
wide_char_constant utf8_char_constant utf16_char_constant
utf32_char_constant string_literal wide_string_literal header_name
utf8_string_literal utf16_string_literal utf32_string_literal l_square
r_square l_paren r_paren l_brace r_brace period ellipsis amp ampamp
ampequal star starequal plus plusplus plusequal minus arrow minusminus
minusequal tilde exclaim exclaimequal slash slashequal percent
percentequal less lessless lessequal lesslessequal spaceship greater
greatergreater greaterequal greatergreaterequal caret caretequal pipe
pipepipe pipeequal question colon semi equal equalequal comma hash
hashhash hashat periodstar arrowstar coloncolon at lesslessless
greatergreatergreater caretcaret auto break case char const continue
default do double else enum extern float for goto if inline int _ExtInt
long register restrict return short signed sizeof static struct switch
typedef union unsigned void volatile while _Alignas _Alignof _Atomic
_Bool _Complex _Generic _Imaginary _Noreturn _Static_assert _Thread_local
__func__ __objc_yes __objc_no asm bool catch class const_cast delete
dynamic_cast explicit export false friend mutable namespace new operator
private protected public reinterpret_cast static_cast template this throw
true try typename typeid using virtual wchar_t alignas char16_t char32_t
constexpr decltype noexcept nullptr static_assert thread_local memchr
memcmp memcpy memmove memset strcat strchr strcmp strcoll strcpy strcspn
strerror strlen strncat strncmp strncpy strpbrk strrchr strspn strstr
strtok strxfrm clearerr fclose feof ferror fflush fgetc fgetpos fgets
fopen fprintf fputc fputs fread freopen fscanf fseek fsetpos ftell fwrite
getc getchar gets perror printf putc putchar puts remove rename rewind
scanf setbuf setvbuf sprintf sscanf tmpfile tmpnam ungetc vfprintf
vprintf vsprintf NULL size_t wchar_t abort abs atexit atof atoi atol
bsearch calloc div exit free getenv labs ldiv malloc mblen mbstowcs
mbtowc qsort rand realloc srand strtod strtol strtoll strtoul system
wcstombs wctomb hash_if hash_endif hash_ifdef hash_elif hash_ifndef
hash_else hash_define hash_undef hash_error hash_include hash_line strdup
strndup snprintf vsnprintf null bufsize len fd dest stdin stdout stderr
EOF TRUE FALSE FILE cin cout endl u32(u32 uint32_t UINT32 uint32 DWORD)
i32(int32_t INT32 int32 INT) buffer(buf buff buffer)
```

2. All Evaluation Metrics for RF-CNN-3

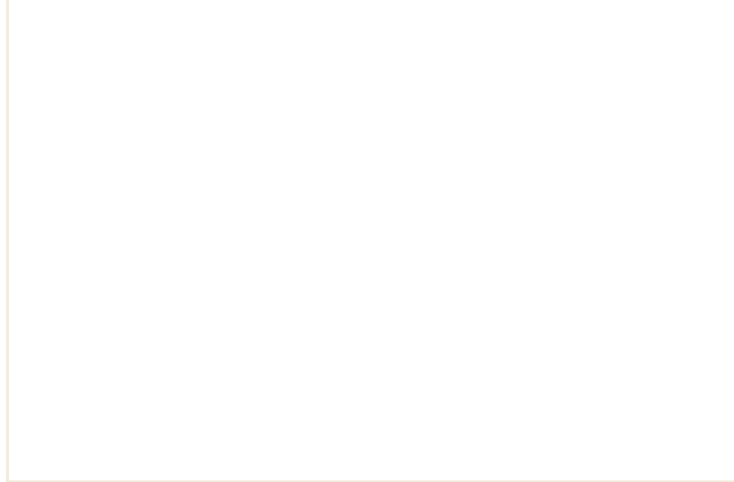
This is appendix contains all metrics calculated for RF-CNN-3 and RF-CNN-3-Binary using 2500 estimators.

RF-CNN-3

```
Thresholds: [0.2847, 0.2928, 0.1295, 0.1172, 0.2042]

 CWE-119
[[122903   2064]
 [   529   1923]]
ACC: 97.965
AUC ROC: 0.961
AUC PR: 0.546
MCC: 0.606
F1: 0.597

 CWE-120
[[118434   4094]
 [  1056   3835]]
ACC: 95.958
AUC ROC: 0.955
AUC PR: 0.557
MCC: 0.597
F1: 0.598

 CWE-469
[[126925    216]
 [   181     97]]
ACC: 99.688
AUC ROC: 0.964
AUC PR: 0.244
MCC: 0.327
F1: 0.328

 CWE-476
[[125915    312]
 [   584    608]]
ACC: 99.297
AUC ROC: 0.912
AUC PR: 0.527
MCC: 0.577
F1: 0.576

 CWE-other
[[121948   1981]
 [  1573   1917]]
ACC: 97.211
AUC ROC: 0.918
AUC PR: 0.470
MCC: 0.505
F1: 0.519

 Overall
AUC ROC: 0.942
AUC ROC weighted: 0.942
AUC PR: 0.468
AUC PR weighted: 0.520
MCC: 0.523
F1: 0.524
ACC: 98.024
```

RF-CNN-3-Binary

```
Thresholds: [0.25268]

AUC ROC: 0.921
AUC PR: 0.568
MCC: 0.567
F1: 0.591
ACC: 93.684
```



NTNU
Norwegian University of
Science and Technology