UiO : Department of Informatics
University of Oslo

Vulnerability Detection using Recurrent Neural Networks

A novel approach towards software vulnerability detection

William Arild Dahl

Master's Thesis Spring 2020

Vulnerability Detection using Recurrent Neural Networks

William Arild Dahl

Spring 2020

Acknowledgements

I would like to express my sincere appreciation to my supervisors. It has been a great pleasure to work with Researcher László Erdődi and Postdoctoral Fellow Fabio Massimo Zennaro. They have provided invaluable counsel from their expertise, and I wish to thank them for that. I would also like to thank the staff at UNINETT Sigma2 for providing us with HPC resources and necessary user support.

Abstract

Information security needs to ensure the safety of an increasingly complex and connected technology structure. The demand for robust and secure IT systems is more critical than ever before. Early identification of vulnerabilities in software is an important measure to meet the high and justified security standards for modern-day information systems. This master thesis investigates the possibilities and limitations of applying a celebrated machine learning paradigm to software vulnerability detection.

We propose a novel approach by applying Supervised Learning using a Recurrent Neural Network constructed by layers of Long Short Term Memory cells. We have limited our research to one of the most relevant and arguably most dangerous class of vulnerabilities, the Stack-Based Buffer Overflow. We have carefully generated and engineered data for training and testing in order to replicate software in the "real world".

We run several experiments using our data and our models in order to validate specific hypotheses about the neural network. We designed each experiment with the intent to unveil whether or not necessary attributes for the neural network are achievable. E.g., accuracy, effectiveness, and sensitivity to low variance data.

Our research indicates that there is indeed some merit to our approach. Even with a "shallow" neural network architecture, our models were able

to perform surprisingly well. As data complexity increased, so did the required training time for our models before converging to a sufficient level of accuracy. Our research also indicates that for complex data distributions, hyperparameter optimization is increasingly challenging.

We conclude that Recurrent Neural Networks are capable of and succeed in differentiating between benign and vulnerable software. Our research has several limitations and a narrow scope. However, it aims to determine a proof of concept. Thus, we believe that our conclusion holds. Further research is needed to discover the potential of using machine learning as an accommodating tool for discovering vulnerabilities in software.

All findings are restricted to our environment and the limitations of our study. We do not attempt to extrapolate results beyond this exact scope.

Table of Content

Table of Content 4

1 Introduction 7

1.1 Area of research 8
1.2 Motivation 8
1.3 Thesis objective 10
1.4 Research methodology 11
1.5 Implementation overview 12
1.6 Related work 13
1.7 Structure 14

2 Theoretical background 17

2.1 Stack Based Buffer Overflow 18
2.1.1 History & relevance 18
2.1.2 Buffer overflow explanation 19
2.1.3 Sample code & execution 21
2.2 Bottom-Up Perspective on a Neural Network 24
2.2.1 The Neuron 24
2.2.2 Activation functions 26
2.2.3 Neural network architecture and deep learning 28
2.2.4 Loss function and propagation of error 29
2.2.5 Gradient decent and learning through optimization . . 32
2.2.6 Regularization 35
2.3 Introduction and History of Deep Learning 36
2.3.1 Supervised Learning 38
2.3.2 Unsupervised learning 39

2.3.3 Reinforcement learning 40
2.4 Recurrent neural networks 41
2.4.1 Long Short Term Memory (LSTM) 44
2.4.2 Gated Recurrent Unit (GRU) 46
2.4.3 Why we choose RNN's 47
2.5 Summary 48

3 Literature review and taxonomy 49
3.1 Static features / standard ML approaches 53
3.2 Static features, convolutional networks 57
3.3 Static features RNN 58
3.4 Dynamic features standard ML approaches 60
3.5 Summary 63

4 Challenges and Limitations 64
4.1 Computational limitations 65
4.2 Data limitations 65
4.3 Classification limitations 66
4.4 Summary 66

5 Approach and Implementation 67
5.1 Data Generation 68
5.1.1 Web scraping 68
5.1.2 Generating & creating data 71
5.1.3 Testing vulnerable & benign code 76
5.1.4 Generator script 77
5.1.5 Initial ideas for choosing functions and structuring them 81
5.2 Data representation & cleaning 85
5.2.1 Representational level 86
5.2.2 Cleaning data samples: 88
5.3 Architecture overview 96
5.4 Summary 104

6 Results and Evaluation 105

6.1 The network can classify a single class 109
6.2 The network can detect a single vulnerability 111
6.3 The network can detect multiple vulnerabilities 114
6.4 The network can differentiate between vulnerable and benign counterparts 117
6.5 Quantitative experiment 120
6.6 Summary 125

7 Further research **127**

8 Conclusion **131**
8.1 Final thoughts and inspirations 134

Bibliography **135**

List of Figures **143**

List of Equations **146**

Appendix A Appendix **148**

Chapter 1

Introduction

In this chapter, we will provide a short description of the area of research, the research problem, and our novel approach towards it. We are to present the underlying issues and challenges which sparked our interest in using classical deep learning techniques for vulnerability detection. We will also introduce the objective of our thesis as well as our motivation grounded in these challenges. In the last section of this chapter, we will lay out an overview of our dissertation to ease further reading and provide a general overview of our work.

1.1 Area of research

This master's thesis spans two major research fields: Information Security (Infosec) and Machine Learning (ML). The union of these two areas has become an important research subject as information security has become more relevant than ever before. We, as well as other researches, are exploring the possibilities and limitations of extending and supporting information security through machine learning techniques.

The ISO/IEC definition of information security is "Preservation of confidentiality, integrity, and availability of information" [1], which we believe encompasses the core of information security in an excellent way. Infosec is a set of practices intended to keep data secure from unauthorized access or alterations. One of these practices is vulnerability assessment, which involves defining, identifying, and classifying security holes in information technology systems. In this master thesis, we are concerned and focused on the **identification** or **detection** of such vulnerabilities in software.

Machine learning is a field of research that is continually evolving and is in rapid change. Historically we say that Machine Learning is a subset of Artificial Intelligence (AI) that has revolutionized several fields in the last few decades. We would like to mention Deep Learning (DL) as well, which is a branch and broader family of machine learning methods based on artificial neural networks. We will elaborate on neural networks, machine learning, and deep learning in our theoretical background chapter.

1.2 Motivation

Our motivation comes from two standpoints: an information security standpoint and a machine learning standpoint. We will start with our motivation concerning information security.

One of the many ways to improve software security is to identify and repair vulnerabilities. Flaws in code is a multi-billion dollar expense yearly. Malware developed to exploit these vulnerabilities has become an ever-increasing

problem for users, corporations, and governments worldwide. WannaCry[2], which is, if not one of the most relevant and dangerous ransomware in recent history, emerged from a Server Message Block (SMB) vulnerability. The EternalBlue [3] exploit utilized this vulnerability. When the Shadow Brokers [4] leaked the exploit, it gave rise to the WanaCry ransomware, which alone ended up costing an estimate of 8 billion USD globally.

Given today's business priorities, developers are creating software at a fast pace, which consequently introduces flaws and bugs in code. The number of vulnerabilities discovered each year has steadily increased over the last decades, with a record of 16,665 discoveries in 2018. [5] Even though this could be interpreted as a positive trend, we argue that the overall risk has increased. The impact of a security breach has dramatically increased because applications are the custodians of more critical data and functions than ever before.

Software vulnerabilities can stay undiscovered for a long time before they are detected. Detection usually happens in one of four ways:

1. **Accidentally**: A use case appears that crash an application.
2. **Fuzzing:** Providing invalidate data on purpose.
3. **Analyzing malware:** Reverse engineering malware may give reveal a used unknown vulnerability.
4. **Analyzing software:** Scrutinizing software with static or dynamic code analyzers such as Google CodeSearchDiggity [6], Joern [7] or Flawfinder [8].

Given the ever-increasing complexity of software structures and its elevated integration across infrastructure, undiscovered vulnerabilities may lead to detrimental consequences. As security researchers, we are highly motivated by the importance of scrutinizing potential additions to the field of study.

From a machine learning standpoint, we are interested in the possibilities of treating assembly code as a language. For an extended time, Recurrent

Neural Networks(which is a class of neural networks) have been the state-of-the-art neural network class for processing sequences of data. Whether it be text, time-series predictions, handwriting, or speech recognition, RNNs have proven success in exhibiting temporal dynamic behavior. Our work provides an excellent opportunity to assess the use of an unconventional class of data in recurrent neural networks.

Even though our undertaking is complex, we are exceedingly motivated to contribute to an exciting and evolving bailiwick.

1.3 Thesis objective

This master thesis aims to determine the feasibility and limitations of vulnerability detection on binary executables by utilizing a supervised machine learning technique. We chose to use a Recurrent Neural Network with Long Short Term Memory(LSTM) cells. This class of artificial neural networks has an excellent ability to handle input vectors of arbitrary and non-uniform lengths. RNNs are also well suited for extrapolating context in sequences of data. RNNs with LSTM cells have proven effective in tasks related to natural language.

Code or programming languages share some fundamental similarities with natural languages. Both have syntax, semantics, pragmatics (they are heavily context-dependent), and grammar. Whereas natural language conveys information between humans, code is concerned with configuring and stating what a computer should do. These similarities fit our desire to explore the possibilities of treating assembly code as a natural language and apply a tried and true machine learning paradigm. There are apparent differences between code and natural languages, such as simpler syntax or less uncertain and diverse pragmatics. However, we still believe that the underlying structure of code is similar enough to natural languages, such that our approach is worth exploring.

Our research elaborates to prove or disprove whether or not binary classi-

fication on code files, where the goal is to differentiate between benign and vulnerable files, is feasible. In turn, the results will allow us to make a statement about the viability and practicality of our novel approach toward vulnerability detection.

We expect to unveil the challenges of our novel approach, draw a conclusion based on our hypothesis, and discover possible extensions of our work through our research. We acknowledge that our objective is exceedingly difficult to achieve. Thus, we would like to state that any negative results can be of use in revealing or disclosing the potential or limitation for parts of or the entirety of our particular approach. In itself, this information could be useful for guiding future research in the right direction.

1.4 Research methodology

This section will serve the purpose of a high-level overview of our research method. Further details and elaborations about each step of our research methodology recommence in their corresponding chapters. Details include alternative and available options, the reasoning behind our choices, and run-through our development pipeline.

We started our research by making a hypothesis; Can Recurrent Neural Networks detect software vulnerabilities in code files? Our hypothesis is extensive. To fit the scope of a master's thesis, we narrowed our research down to exploring the limitations and possibilities of our approach with a single class of vulnerabilities.

To evaluate and determine whether our approach of using supervised learning for vulnerability detection was feasible or not, we set up a series of experiments. We devised each experiment with the intent of unveiling the potential or limitation of our approach gradually. For each experiment conducted, we perform an individual evaluation. This way, we can underpin and establish a basis for an overall evaluation, and eventually draw a conclusion.

We decided to assess our hypothesis in an artificial setup/environment by

developing a collection of functions divided into two libraries of vulnerable and benign functions. We used these functions to generate fully functional and executable code files. In turn, these code files are compiled and transformed into data samples for training and testing our neural network model. This approach gives us greater control of the data samples concerning size, complexity, and variance.

We chose to develop our code for the recurrent neural network by using PyTorch [9], an open-source machine learning library for Python. By designing the architecture, configurations, and developing our neural network ourselves, we were able to tailor the network to our specific needs. In turn, by creating everything ourselves, we achieved a deeper understanding of our neural network's inner workings and behavior on different configurations and distributions of training data.

1.5 Implementation overview

In order to conduct our research and test our hypothesis, as described in the methodology and thesis objective section, we had do develop several tailored pieces of software. In order to run our experiments, we developed the following programs:

Function libraries: Two libraries that contain benign and vulnerable functions written in C language code. All vulnerabilities contain stack-based buffer overflow flaws. Half of the benign library consists of repaired functions from the vulnerability library.

Generator: A python program which generates fully functional program files written in C code. This generator program, in conjunction with the function libraries, allowed us to generate thousands upon thousands of unique program files. Eventually, these program files are used to create the datasets we use for our experiments.

Compiler: A python program that compiles all the generated C programs into assembly programs. We use the GCC compiler with Intel

style syntax. The compiler was crucial as we were often compiling thousands of binaries at the time.

Data cleaner: A python program that does the bulk of our feature engineering. Assembly files are labeled, stripped of superfluous information, and written to a training and a testing file. These two files are input to our neural network.

Neural Network: A python program that consists of all the code for our neural network implementation. This neural network allows us to run our experiments various configurations, such as hyperparameters and training and test data.

Further details, explanations, and code snippets on our software are included in chapter five. All code associated with our thesis is available at the following git repository [10]. A thorough guide on how to perform our experiments will be provided on this repository.

1.6 Related work

The area of research on vulnerability detection using ML is still uncharted, so no absolute solution has been determined or discovered yet. Many of the relevant papers are concerned with malware detection, which is dissimilar to what we opt to do. However, there are promising investigations and indeed successful implementations in using machine learning techniques for vulnerability detection. We will elaborate further on the state of the field of research in our literature review. Our approach is, in many ways, different from the previous work done. This concerns both data, features, representational level, and model architecture. We thoroughly elaborate on decisions and choices in their respective chapters, but we would like to present an overview of the main dissimilarities:

Data: To begin with, we generate our own data. Other research collected samples from various sources. We argue for the sake of our objective that the generated data achieve its purpose.

Features: Like most related research, we are also utilizing static features. As we are treating code as a language, this is the natural approach. We do not analyze any dynamic features as we would like our model to be able to do vulnerability detection on "cold" binaries, that is, not executed.

Representational level: Our approach is unique with regard to representational level. While other research on static features assumes available source code, we do not. One objective for our hypothesis is that we would like to analyze binaries where source code is not available. However, the assembly instructions we use can be derived by using various decompilers. In chapter five, we conduct further discussions around the representational level.

Model: Whereas most other approaches assemble multi-layered deep neural networks in conjunction with or utilizing other advanced machine learning paradigms. We choose a simplistic and minimalist approach.

Since the purpose and goal for our master thesis is to research a new and unexplored approach towards vulnerability detection, it is natural for us to deviate from the already traversed propositions.

1.7 Structure

Our thesis is divided into eight chapters, including this introduction. The introductory first chapter has served as a primer and an overview of the main aspects of our work. It has offered an introduction to our fields of research, the motivations behind our research, and stated the research objective. The chapter has also given a high-level explanation of methodology, implementation, and a comparison with related work.

Theoretical focus and underpinnings can be found in chapter two. We present four central areas in which we believe the reader should be familiar. These are, in order: the stack-based buffer overflow vulnerability, a bottom-up per-

spective on neural networks, an introduction to deep learning, and ultimately recurrent neural networks. In this chapter, we also provide our reasoning behind using Long Short Term Memory cells in our neural network.

Related sources and taxonomy of our field of research is presented in chapter three. In this chapter, we provide a comprehensive overview of how far the field of research has advanced by reviewing the most relevant papers for our objective. This chapter also functions as a guideline for evaluating our findings.

The scope of a master thesis has its boundaries and limitations. In the fourth chapter, we state our challenges and limitations. They have, in turn, guided the scope and the design of the thesis objective. We cannot possibly hope to cover everything, and with our limited resources, we had to make several compromises. These compromises span vulnerabilities, data gathering, and model architecture.

In chapter five, we present our approach and implementation for generating, compiling, and feature engineering our data. To make this process as intuitive as possible, we have included a pipeline-like diagram. This chapter also encompasses an overview of our developed model architecture, code samples, and discussions around our choices to the aforementioned content.

The core of the thesis are the experiments conducted. We present all experiments, the goal for each of them, their configurations, and datasets in chapter six. We document the results and individual evaluation with a conclusion about whether or not the experiment was successful. At the end of the chapter, we make a generalized evaluation and assessment of the experiments and what we can interpret from them.

Concerning our limitation and scope, we have many interesting possible extensions of our work. The received results have also raised additional questions and challenges. In chapter seven we discuss these potential expansions. We hope that these challenges might inspire forthcoming researchers and help push the field of research further.

In the last chapter, we summarize the problem, our main findings, and make

a statement on whether or not our approach holds merit towards our problem statement. We also take a critical standpoint towards our findings and consolidate our limitations, research results, and evaluation of experiments in order to establish a well-grounded conclusion. Finally, we conclude the chapter with some final thoughts and inspirations for professionals in the field of research.

Chapter 2

Theoretical background

This chapter aims to provide the reader with the theoretical foundations of our dissertation. We are to present three central domains of knowledge that we believe would benefit the reader. In order, we will establish sufficient knowledge about stack-based buffer overflows, neural networks, and finally, deep learning with a particular focus on recurrent neural networks.

2.1 Stack Based Buffer Overflow

This section of the thesis will provide a brief historical background of the buffer overflow weakness, a technical explanation, and the motivation for choosing buffer overflow as a starting point of our vulnerability detection.

We will begin by limiting our model to do training on binaries, either containing or not containing a buffer overflow weakness. Buffer overflow is a bug that allows writing outside the boundaries of allocated memory, which can corrupt data, crash the program or cause execution of a crafted payload by taking control of the stack and base pointer. This is enabled by for example misusing various string functions and vulnerable API's in C / C++ i.e **strcpy()**, **gets()**, **scanf()** and so on. In general, we separate buffer overflows into two main categories; either overflowing the stack (Stack overflow) or overflowing the heap (Heap overflow) [11]. Stack-based buffer overflows are more common than heap-based overflows and leverage the stack memory, which exists during a function's execution time. Heap-based attacks are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations. We have chosen to limit ourselves to the former, as it is more common, easier to implement.

2.1.1 History & relevance

Buffer overflows were documented and understood as early as 1972 by the Computer Security Technology Planning Study [12]. The earliest documented exploitation buffer overflow was in 1988, where it was one of several exploits used by The Morris Worm [13]. By type, buffer overflows takes the top spot of the most reported incident in the past 25 years. It is also the highest-ranked category of vulnerabilities concerning "high" or "critical" severity. Buffer overflows were the top vulnerabilities from 1988 - 2005 and have been in the top three ever since [14].

Two common defences against buffer overflow attacks are Address space layout randomization **(ASLR)** and using Data Execution Prevention **(DEP)**.

ASLR is a defence mechanism in which the locations of system executables are randomized as they load into memory. DEP prevents applications from executing code from a non-executable memory location. However, even with these and other mitigation strategies, buffer overflows are highly relevant to this day. ASLR and DEP are strong security measures, but some exploits can bypass each of them [15]. It is also worth noting that even though we utilize these security measures, they only alter the bug's consequences. The application might still crash, which is severe enough in itself. Another reason why buffer overflows are still relevant is that we still write and probably will continue to write low-level code (which is more susceptible to stack-based exploits) for our hardware interfaces and OS libraries.

2.1.2 Buffer overflow explanation

With the relevance and history of the stack-based buffer overflow established, we would like to provide a general understanding of how stack-based buffer overflows work. First of all, let us take a look at a simplified memory layout of a Win32 distribution.

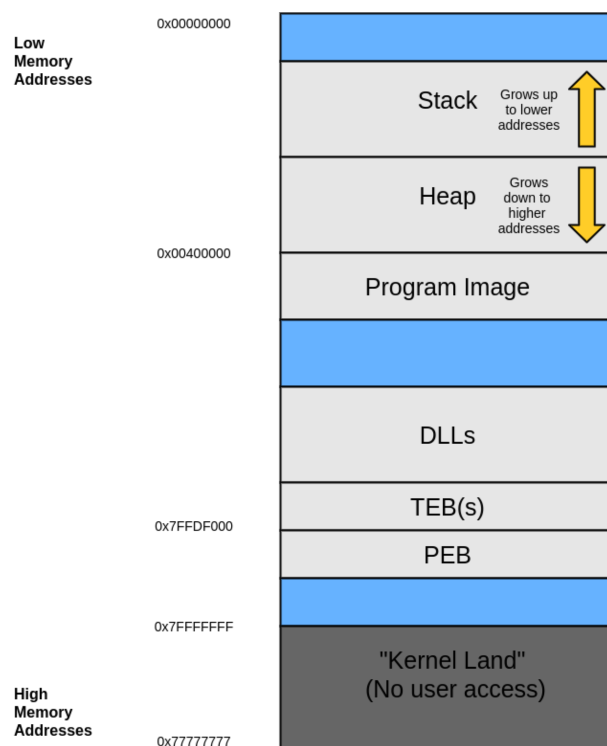

Figure 2.1: Memory Map

From figure 2.1, provides a rough sketch of how memory is partitioned after a program loads in memory. We can also observe how these partitions spread across different addresses. We will focus primarily on the **stack**, which contains essential variables and addresses used by executables during runtime, for example, return address, local variables, and function variables. Without ASLR, it is relatively easy to deduce where a particular buffer will reside on the stack. A **buffer** in this context is an implementation of a variable with a continuous segment of memory allocated to it. It can be of variable length, depending on what the programmer needs. An example could be a buffer used to store some user input. If the value assigned to a variable exceeds the buffer allocated to it, memory locations not allocated to the variable might be overwritten (we will present an example shortly). Something else that is important to understand is that, if this overwritten memory location was allocated to another variable, that other variables value is overwritten.

These two traits of the stack are what make a stack-based buffer overflow possible.

f an entity controls the stack with malicious intent, it is easy for the entity to cause severe harm. To provide a more in-depth understanding, we will examine the system stack and how it operates. When a function call is made, the stack creates a **stack frame**. This stack frame contains all function arguments, return address (location were the function jumps back when program flow is resumed), and all the statically allocated buffers. When the function arrives at the return address, execution continues at that specified address. Whatever instruction is being pointed at will be executed by the CPU. The CPU has many registers, but we will focus on the ones important for us;

EIP: Extended Instruction Pointer
ESP: Extended Stack Pointer
EBP: Extended Base Pointer (frame pointer)

Figure 2 provides an illustration on how a stack frame is layed out:

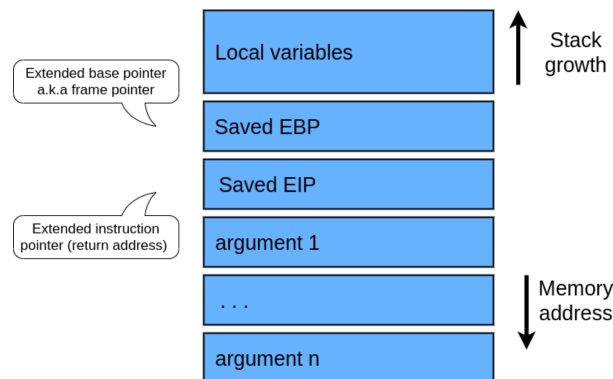

Figure 2.2: Stack Frame

2.1.3 Sample code & execution

Here is a small example program that expects input from a user. It has allocated 12 bytes of memory to the variable **buf**, and it will read and copy whatever value the user provides into **buf**.

```c
void copyBuf (char* str){
    char buf[12];
    strcpy(buf, str);
}

int main (int argc, char** argv){
    copyBuf(argv[1]);
    return 0;
}
```

Listing 2.1: Buffer overflow vulnerable example

If the user provides input within the size limits allocated to our variable, all is fine and good. However, if a user, for example, provides a lengthy input, say a series of 24 A's (see figure 2.4), which is 12 bytes more than expected, we get a situation where the saved frame pointer, the return address, and address to argv[1] are overwritten.

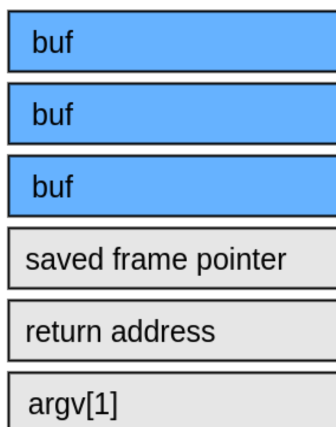

Figure 2.3: Safe usage of buf

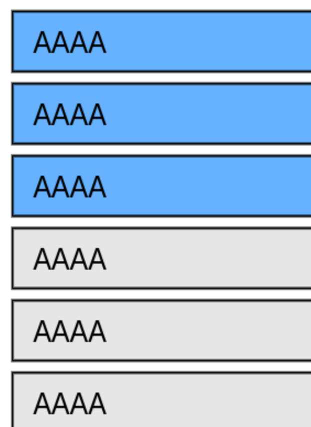

Figure 2.4: Overwritten

If someone is able to overwrite the return address, they control the EIP when the function returns, which means that they can alter the program flow. What a malicious entity can do is to fill the buffer with commands and redirect program execution and we end up with a stack frame looking something like figure 2.5:

Figure 2.5: Malicious Buffer Overflow

As one can imagine, these types of attacks are hazardous, as this malicious code, often called **payload**, will be executed with the same privilege as the original program. Our goal for this thesis is not too concerned with the actual malicious parts of a stack-based buffer overflow, but rather if it is possible to detect such a weakness in the program code.

With the relevance of the vulnerability established, the second part of the motivation using buffer overflows in the training data is its ease of implementation. With a single method and just a few lines of code, we can create an entirely "functional" buffer overflow vulnerability. It is also one of the easier to understand and most taught vulnerability that exists to this day. Another reason is that it is also straightforward to modify these vulnerable functions and make them benign, which we also need in our training data. Therefore, buffer overflows are an obvious starting point for our research with respect to relevance and ease of implementation.

2.2 Bottom-Up Perspective on a Neural Network

In this section, we present and explain the inner workings and basic logic parts of a neural network. We would like to explain the foundations of neural networks and the architectures that are relevant to this dissertation.

2.2.1 The Neuron

Historically, neural networks were inspired by modeling biological neural systems. The first breakthrough happened in 1943 when Warren S. McCulloch and Walter H. Pitts successfully modeled an artificial neuron as a mathematical function [16]. Neurons are the basic computational units of the brain. Each neuron is connected with other neighbour neurons through **synapses**. Each neuron also receives input signals from its **dendrites** and produces output signals through its **axon**. The axon then branches out and connects to other dendrites through synapses of other neurons [17].

Figure (2.6) and (2.7) shows simplified images of a biological neuron, and its mathematical model respectively. From the image, we can observe the mathematical representation of each basic part of a neuron. We have a signal that travels along the axon of a neuron, e.g., x_0, which interacts multiplicatively, i.e., a dot product $x_0 w_0$ with the dendrites, which creates an input signal for the receiving neuron based on the strength of the synapse w_0. The strength of the synapses between neighbour neurons is where learning happens, both in the "real world" and in the mathematical model. Each neuron has a set of n synapses, which we refer to as **weights** from now on. The weights can be both positive and negative and influence the neuron in an excitatory or inhibitory manner. In the basic model below, the dendrites carry the input signals to the neuron, where they are summed up. If the sum of inputs is above some threshold, the neuron can **fire** a signal along its axon. The signals are simplified in the mathematical model, where we only care about its strength. The amplitude of the neuron's output signal is calculated by

a pre-defined **activation function** f. In addition, each neuron has a **bias** b, which acts as a constant value to the activation function to allow better flexibility and predictions for a model [18]. There exist several different activation functions, which all have their strengths and weaknesses. However, historically the **sigmoid function** $\sigma(x)$ is the one most used as it approximates a simple step function and can be easily derived. We will go into more detail about both the sigmoid, and the **tanh function** $tanh(x)$, which are two commonly used activation functions in recurrent neural networks.

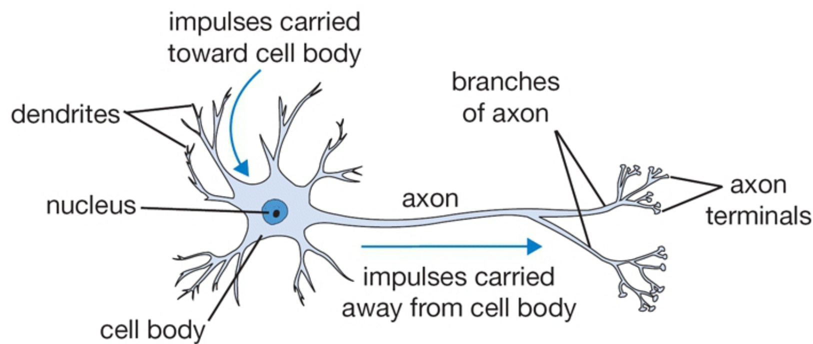

Figure 2.6: Simplified illustration of a neuron

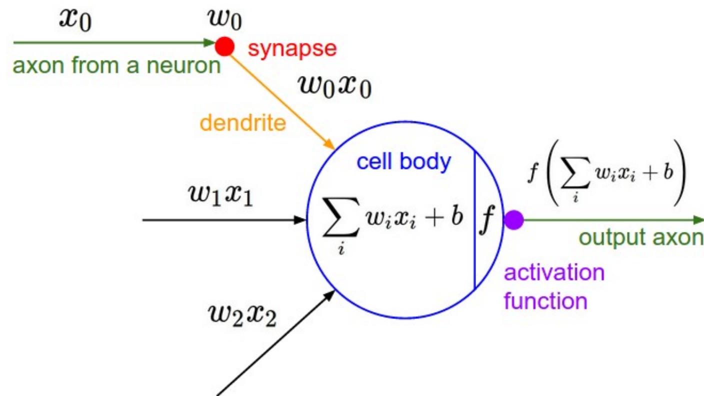

Figure 2.7: Neuron represented as a simplified mathematical model

As we can observe, the mathematical model is quite simplified compared to the real neuron of a brain. However, it is sufficient at representing and providing a basic building block of a neural network. In short, each neuron

performs a dot product with its inputs and weights, adds a bias, and eventually applies a non-linear activation function to determine output.

2.2.2 Activation functions

Every activation function takes in a single number input and performs a non-linear mathematical operation on it. As mentioned, we have many different activation functions, which all have their place in machine learning and deep learning. However, we will focus on two of the most relevant for our problem. Conveniently they are the two most common activation functions. We will not elaborate on the pros and cons of each of them, but describe them briefly and present the mathematical formula for both.

Sigmoid or Logistic Activation Function

The sigmoid non-linearity takes a real-valued number and "squashes" it into a range between 0 and 1. This way, large negative numbers approaches 0, and large positive numbers approach 1. The sigmoid function is, as mentioned, one of the most used activation functions, but has recently fallen out of favor. However, as we will see in the next section, it plays a major role in the neurons of a recurrent neural network. Mathematically the sigmoid function has the following form:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

If we take a look at graph, (which shows us the sigmoid non-linearity) we can see that sigmoid has an S-curve around 0.5, with both ends approaching 0 and 1 for values approaching infinity.

Tanh / hyperbolic tangent Activation Function:

Tanh is also a non-linearity which also takes in a real-valued number. However, tanh is zero-centered, and "squashes" the value to the range [-1,1]. It

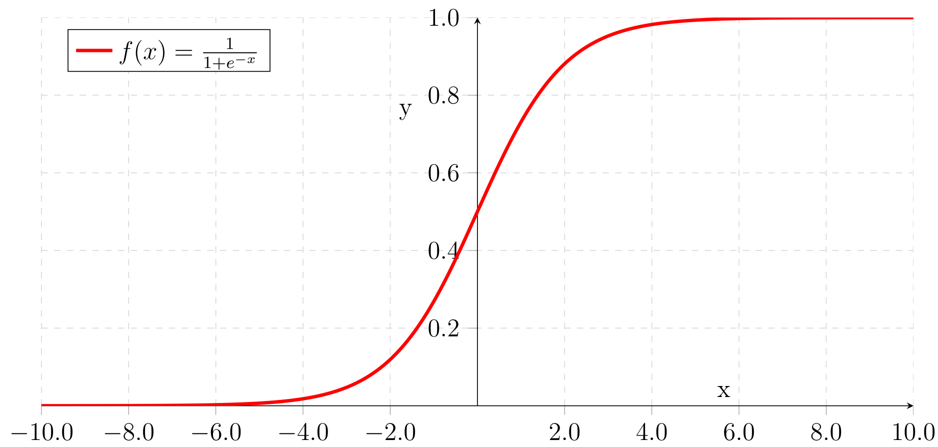

Figure 2.8: The sigmoid activation function

is usually prefered over sigmoid due to its zero-center and is a crucial part of recurrent cells. From the graph illustration below, we can observe that tanh and sigmoid are quite alike. Tanh looks simply as a scaled version of the sigmoid function. Mathematically, the following holds:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.2}$$

As we can observe, the tanh function is an extension of the previously mentioned sigmoid activation function. On a side-note, some other commonly used activation functions include: the identity function, The Rectified Linear Unit (ReLU), leaky ReLU, Exponential Linear Unit (ELU) and SoftPlus [19].

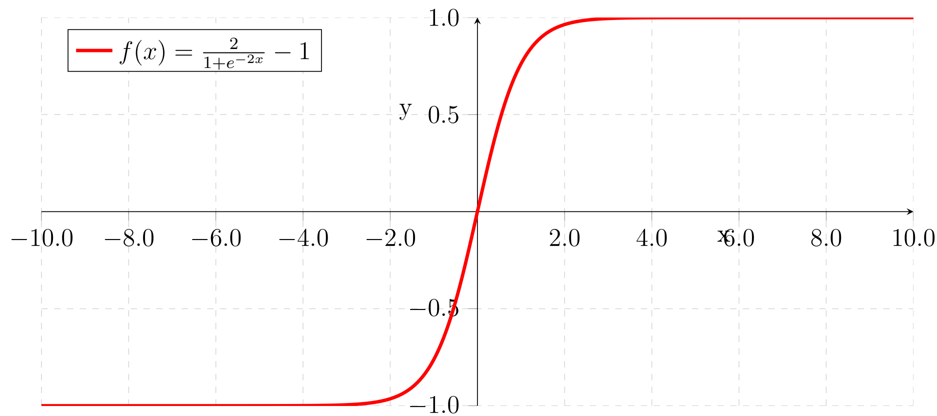

Figure 2.9: The tanh activation function

2.2.3 Neural network architecture and deep learning

With the smallest building blocks established, we can take a step back, and look at how these neurons and weights compose a neural network. For simplicity's sake, we will start by presenting a vanilla neural network before we go into the type of architecture we will use for this thesis. The neurons of a neural network are connected layerwise as an (in most cases, except recurrent networks) acyclic graph. The most common vanilla architecture is a **fully connected** network in which neurons between adjacent layers are fully pairwise connected. The output from one layer of neurons is input to the next layer, and so on until the last layer. It is convention to call the first layer of neurons: **input layer**, the middle ones as: **hidden layers**, and the last layer: **output layer**. The networks "prediction" or rather its computation is output from the last layer.

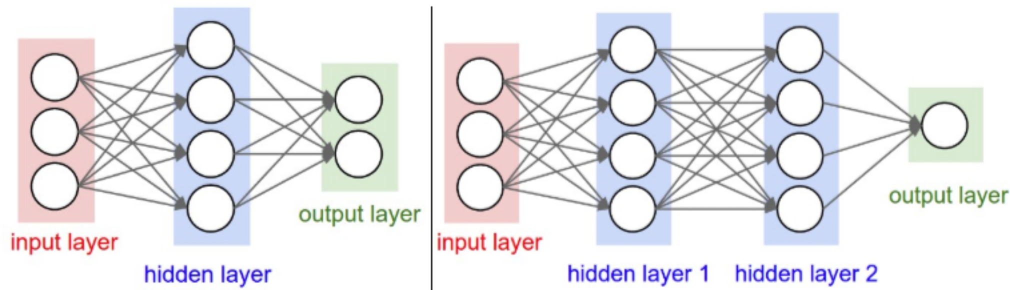

Figure 2.10: Simplified model of two Neural Network. [20] **Left:** Two layer fully connected Neural Network. **Right:** Three layer fully connected Neural Network.

Mathematically we say that: $a_k^{[l]}$ is the activation of node k in layer l.

$$a_k^{[l]} = g(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}) \tag{2.3}$$

Where $w_{jk}^{[l]}$ are the weights between neuron j in layer $l-1$ and neuron k in layer l, $b_k^{[l]}$ is the bias of node k in layer l, and g is the non-linear function.

The term **deep learning** refers to the number of layers in an architecture. The architectures above are simplistic compared to the ones used in deep learning, which often consist of numerous layers stacked on top of each other. Some deep learning architectures worth mentioning are: AlexNet[21], VGG Net[22], Google Net[23] and ResNet[24] just to mention a few.

2.2.4 Loss function and propagation of error

As aforementioned briefly, learning happens in the connections or weights between the layers and in the added bias. These weights are represented as matrices of real-value numbers, usually initialized to some small pseudo-random number. (The act of initializing weights is an entire field of study, and in many cases, poor choice of initialization can make or break learning

in a neural net.) When a neural network provided with input, it is **propagated** forward through the network. Input is multiplied element-wise with the value of the weights and transformed by the non-linearity, which is input for the next layer of neurons. This operation is called a **forward propagation**. The last layer will output some arbitrary real-valued number used to represent some class score (e.g., classification). This value compared to "ground truth" (supervised learning), and we apply a loss or error function to rate or determine the correctness of the prediction. One example of an error function is the cross-entropy cost function:

$$\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{k=1}^{n_y} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{2.4}$$

Here the loss between the true output $y^{(i)}$ and predicted output $\hat{y}^{(i)}$for one class is calculated as a sum of errors over all classes n_y. The goal of a neural network, be it vanilla, recurrent, or convolutional, is to minimize a loss function. There are numerous different loss functions, many of which are well suited for specific or more general problems. This error used to update the weights in the network in a backward pass. The update of weights in a backward pass is called **backpropagation** and was, in fact, a huge breakthrough in machine learning, and a crucial part of all deep learning.

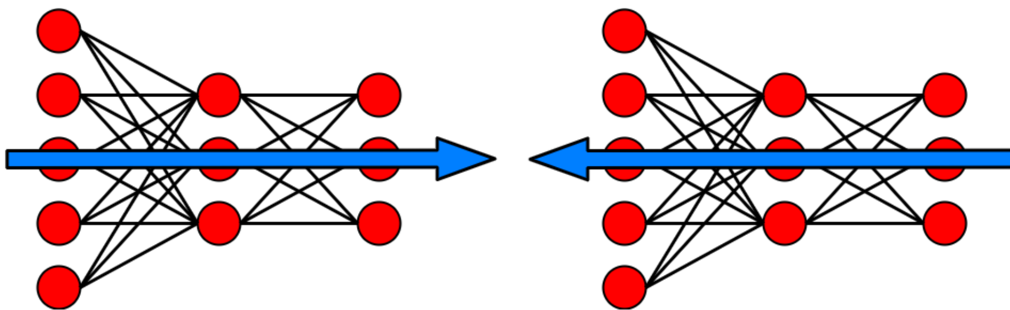

Figure 2.11: Forward and backward pass through a neural network

We would like to explain our chosen loss function now that we have established a fundamental understanding of loss functions and propagation of

error. As stated in the problem statement, we are attempting to perform binary classification. With this in mind, it was natural for us to choose the **Binary Cross-Entropy** (BCE) loss function. BCE efficiently penalizes a neural network for binary classification. Figure (2.12) exhibits the two outcomes of a logarithmic loss calculated over predicted output.

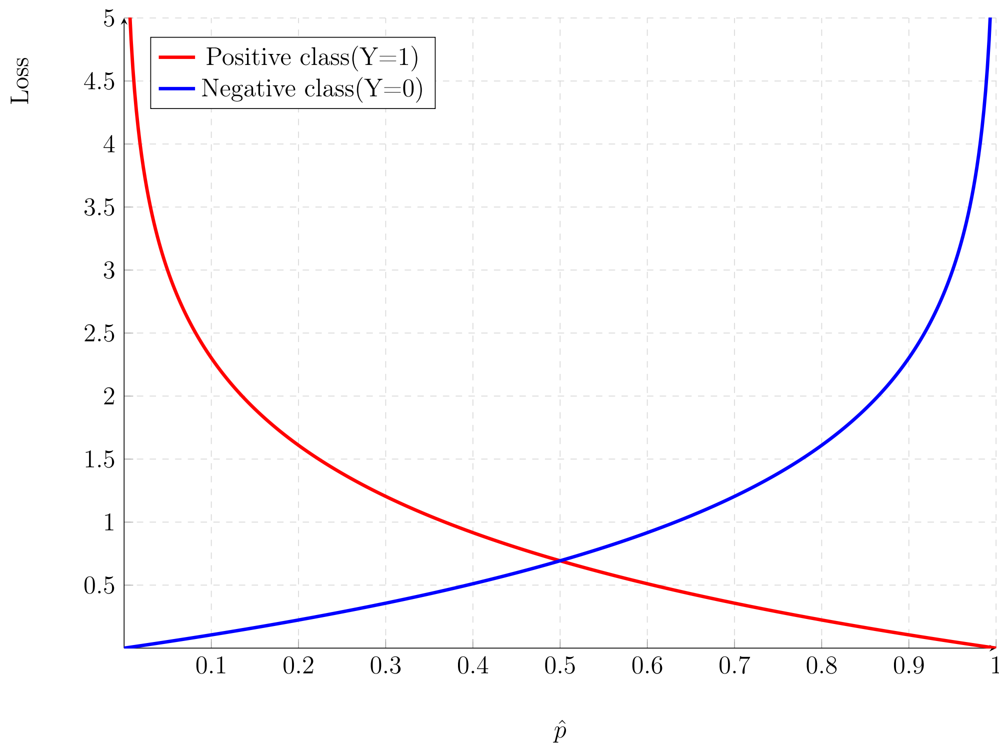

Figure 2.12: Binary Cross-Entropy loss

If we imagine that Positive Class represents vulnerable and Negative Class represents benign we can take a look at how the loss changes following predictions. The plot above gives us a clear picture —as the predicted probability of the true class gets closer to zero, the loss increases exponentially and a exact correct prediction yields a loss equal to zero. This is due to the fact that from a classification perspective, 0 and 1 have to be polar opposites since they each represent completely different classes, i.e whenever the network predicts a sample to class **Y=0 (benign)** when **Y** is **1(vulnerable)**, the loss will have to be very high in order for the network to learn its mistakes

more effectively. This is a desired property of a logarithmic loss function as we want the loss function to return high values for bad predictions and low values for good predictions. Mathematically we express BCE loss as:

$$\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{N}\sum_{k=1}^{N}(\tilde{y}_k^{(i)} \cdot \log(\hat{y}_k^{(i)}) + (1-\tilde{y}_k^{(i)}) \cdot \log(1-\hat{y}_k^{(i)})) \tag{2.5}$$

Where $y^{(i)}$ is the label for a class, and $\hat{y}^{(i)}$ is the predicted probability of a data sample belonging to said class over all samples N. The formula describes that for each vulnerable point (y=1), it adds log(p(y)) to the loss, that is, the log probability of it being vulnerable. Conversely, it adds log(1-p(y)), that is, the log probability of it being benign, for each benign point (y=0). Generally, the BCE loss function is the preferred function for binary classification as it provides an effective calculation of error when the task is differentiating between two classes.

2.2.5 Gradient decent and learning through optimization

Updating the weights and bias happens through an algorithm called **gradient descent**. The goal is then to minimize the error that the trainable parameters are causing. More mathematically, we can say that we want parameters to **converge** to a point where the network can minimize errors for unseen samples of data. Without going too much into details, we can say that:

$$\theta \leftarrow \theta - \lambda\frac{\partial J}{\partial \theta}(\theta) \tag{2.6}$$

Where J is some objective function we want to optimize (i.e., the example of cross-entropy-loss presented above), θ is the parameter we want to update (weight or bias for some neuron k at layer L), and λ is the step length

(often called learning rate in machine learning environments). $\frac{\partial J}{\partial \theta}$ gives us the direction of steepest ascent at point θ_k.

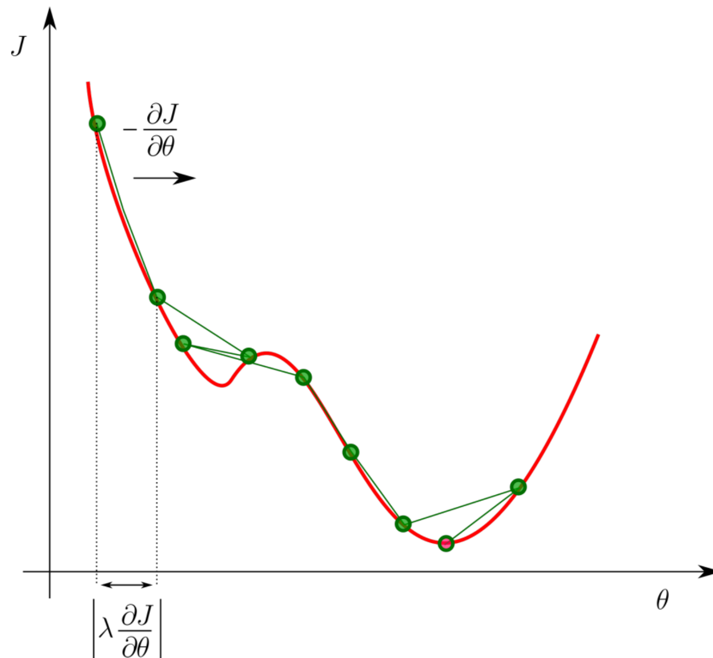

Figure 2.13: Gradient decent on parameter θk [25]

Gradient descent through steepest descent is the most naive, gradient descent optimization method. It is mostly suitable for more straightforward deep learning tasks. As our classification task is very complex, we will have to choose a better optimization algorithm. The choice of an optimization algorithm can potentially shave hours and days of the training process, keeping all other parameters equal [26]. For us, it was only natural to elect the Adam optimization algorithm. Adam is an extension to stochastic gradient descent and has been adopted to a vast number of different deep learning applications, including natural language processing.

Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto presented Adam in their 2015 paper "Adam: A Method for Stochastic Optimization" [27]. Some of Adam's benefits include that it is computationally efficient, has low memory requirements, and is well suited for large-scale

problems or problems with very noisy or spare gradients(the neural network is not receiving strong enough signals to tune its weights). Adam is different from classical stochastic gradient descent by and large in two key properties. First of all, SGD maintains a fixed learning rate for all weight updates throughout the training phase. Adam maintains a learning rate for each network weight and separately adapts it as the training unfolds. The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- **Adaptive Gradient Algorithm:** (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g., natural language and computer vision problems).
- **Root Mean Square Propagation:** (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of current magnitudes of the gradients for the weight (i.e., how quickly it is changing). Thus the algorithm performs well on online and non-stationary problems.

Adam realizes the benefits of both AdaGrad and RMSProp [26]. It is a consensus within the deep learning community that Adam is one of the overall best optimization algorithms. Sebastian Ruder has proven this fact in his comprehensive review of modern gradient descent optimization algorithms[28]. The algorithm has also received an appraisal in the course "CS231n: Convolutional Neural Networks for Visual Recognition" developed by Andrej Karpathy, et al. [29]. As we are exploring binary classification by utilizing natural language processing techniques and process complex data, the Adam optimization algorithm is a great choice for us.

We will not elaborate further on other optimization techniques in this section. Our goal was to establish an understanding of the different parts of a neural network and how they learn through backpropagation.

2.2.6 Regularization

Regularization is a technique that makes slight modifications to the learning algorithm such that the model can better generalize. Which, in turn, improves the model's performance on the unseen data as well. We are not going to describe the numerous different regularization techniques that exist but would like to outline one of them: **dropout**. There is a consensus that deep neural networks are able to extract and build better features than shallow models. They achieve this by using the intermediate hidden layers to build said features from the input data. However, overfitting is a severe problem in such networks. By overfitting, we imply a network that has fitted itself so well to the training data that it will no longer perform well on the development/validation set or any other unseen data. Deep neural networks are also slow to train as the number of calculations increases drastically for each added layer or neuron. Dropout is a technique to address this problem. It is an extremely effective and straightforward regularization technique introduced by Srivastava et al. [30]. The key idea is to drop a neuron and its connections during training randomly. With some probability p, a neuron is kept alive. Otherwise, set to zero.

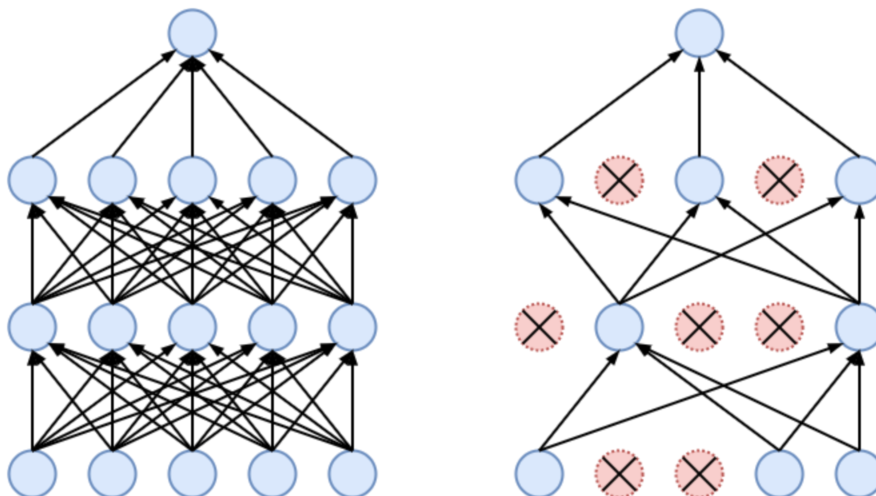

Figure 2.14: Dropout Neural Net Model [30]. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left.

During training, dropout can be interpreted as sampling a neural network within the full neural network and only updating the parameters of the sampled network based on the input data. During testing, there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks. What is important to note is that by using dropout, the output of neurons during test time must be adjusted by the dropout probability. We want the outputs of neurons at test time to be identical to their expected outputs at training time. Consider a dropout p for outputs where $p = 0.5$ for all outputs x. The expected output of neurons would then be $px + (1 - p)0$ as a neuron's output will be set to zero with probability $1 - p$. There are two approaches to adjusting for dropout, either during training or testing. At test time, when we keep the neuron always active, we must adjust $x \to px$ to keep the same expected output. Since test-time performance is critical, it is common to do an inverted dropout that performs output scaling during train time, leaving the forward pass during test-time untouched.

2.3 Introduction and History of Deep Learning

This section will provide a brief introduction and understanding of what deep learning is and cover the major historical events throughout deep learning history. This section will not cover technical details about different historical or present mathematical models used in deep learning. Covering all technical details would quickly fill a whole book and is neither beneficial nor necessary. Technical details about the architecture, model, and techniques we have chosen to use for our problem will be thoroughly elaborated on later in the thesis. Figure 2.15 provides a Venn diagram of the different research fields of artificial intelligence.

We find this introductory section necessary mainly for two reasons. First and foremost, to gain a general understanding of what deep learning is, its

history, and how we have ended up with the state of the art architectures and algorithms used today. Secondly, we think it is essential to give due credit to the scientists and researchers who have pioneered the field and paved the way for future generations. We will start by giving an introduction and provide a general understanding of what machine learning is and how deep learning is related to machine learning. We will now proceed with a brief introduction of the history behind machine learning, and from there, present some benchmarks from deep learning models.

There are many different definitions of Machine Learning. In general, we can say that ML is the scientific study of algorithms and statistical models a computer system uses to learn from experience rather than being explicitly programmed. Stanford's definition of Machine Learning is "Machine learning is the science of getting computers to act without being explicitly programmed" [31].

Generally, we can divide Deep Learning and Machine Learning into three main branches. Supervised, semi-supervised, also known as partially supervised, and unsupervised learning. Additionally, we have a category called Reinforcement Learning (RL), often categorized within the scope of semi-supervised or unsupervised learning. On a side note Spiking Neural Networks (SNN's) is also a brain-inspired paradigm within AI. In SNN, neurons are a function of the width and timing relationships of an input pulse instead of a single value.

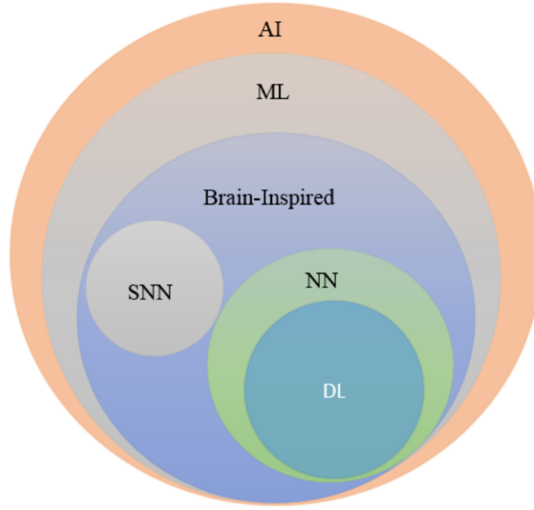

Figure 2.15: Deep learning in context of artificial intelligence. [32]

2.3.1 Supervised Learning

Supervised learning is a learning technique that builds a mathematical model based on labeled data. With a supervised DL approach, a training set of samples containing input and desired "target" outputs is provided training data for the model. During training, the agent predicts $\hat{y}_t = f(x_t)$ and receives a loss value $l(y_t, \hat{y})_t$. The agent will iteratively update its parameters to minimize the given loss function. If the training is successful, the agent should generalize and respond correctly to unseen examples drawn from the same distribution. In mathematical notation, we can say that given a training set Ω with input x and desired output y:

$$\Omega_{train} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\} \tag{2.7}$$

The goal is to create a function f that “approximates” this mapping:

$$f(x) \approx y, \forall (x, y) \in \Omega_{train} \tag{2.8}$$

With the aim that this function f generalizes well to unseen examples:

$$f(x) = \hat{y} \approx y, \forall (x, y) \in \Omega_{test} \tag{2.9}$$

Some of the most used approaches within the supervised learning category include Deep Neural Networks (DNN), Convolutional Neural Network (CNN), and Recurrent Neural Networks (RNN). Supervised Learning algorithms include classification and regression. When there is a fixed amount of output values, e.g., image classification of animals, a classification paradigm is preferred. When the output can have any numerical value within a range, e.g., real estate prices, regression is applied.

2.3.2 Unsupervised learning

Unsupervised learning is quite different from supervised learning, as no labeled training data "exists". Instead, the algorithm tries to identify similarities or structures in the data. This way, data points are clustered or grouped in order to find an underlying structure of the data. To present it in mathematical notation as before, given our training set only consists of input x:

$$\Omega_{train} = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\} \tag{2.10}$$

Since we do not have any targets, unsupervised learning does not apply an error function on each data sample. Instead it utilizes a **clustering** error function to cluster the data. This technique is advantageous when labels for the training data is hard to obtain, or hard to generate. A typical algorithm used in Unsupervised Learning is K-means clustering.

Other popular algorithms used in unsupervised learning are Principal Component Analysis (PCA) and Stochastic Neighbour Embedding. As it is not a

central part of this dissertation, we will not elaborate further on unsupervised learning.

2.3.3 Reinforcement learning

Reinforcement learning is an area of machine learning, where the algorithm or software agent learns by experiences it gains by interacting with an environment. "Labels" are quite different from the ones used in SL, i.e., the agent does not have a "ground truth" about what is the "best" or "correct" solution is. The agent is instead provided with feedback or a reward/penalty for chosen actions and states but never suggestions on improvements. The goal of the agent is to maximize an accumulated reward given by the environment.

$$G_t = R_t + \gamma R_{t+1} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{2.11}$$

Where rewards in later timesteps are penalized by some discount factor γ :

$$\gamma \in [0, 1] \tag{2.12}$$

Due to its generality, typical Reinforcement Learning applications are process controls, networking, robotics resource management, and game theory. Even playing games such as chess or more modern video games, just to mention a few [33].

We will utilize a supervised learning approach for our specific goal as we are using labeled data to train our model to differentiate between classes.

2.4 Recurrent neural networks

Up until now, we have talked about different traditional DL approaches, including DNN's and CNN's. These approaches work well on structured inputs or when we work on images (CNN's). What they are not capable of dealing with is understanding words or sentences based on the understanding of previous words and sentences, i.e., understanding the meaning of something based on a variable context. (One could argue that CNNs recognize images in context.) There are two reasons why DNN's and CNN's are incapable of this. First, and most intuitively, these approaches can only deal with an input vector with a fixed size, and will also produce an output vector of fixed size. Numbers between 0 and 1 are usually representative of probabilities for different classes given a classification problem. The second problem is that DNN's and CNN's have a fixed number of computational steps, which is dependant on the number of layers is the network.

What RNNs allows is the processing of data with unknown length over time. The idea of RNNs is far from new. The concept was first introduced in 1982 in 'The Hopfield Network" by John Hopfield [34]. The basic idea is that a Recurrent Cell creates a new state based on both the new input and the old state of the cell. The cells in a RNN utilize a recurrence formula as follows :

$$h_t = f_W(h_{t-1}, x_t) \tag{2.13}$$

Where h_t is the new state, f_W is some function with parameters W, h_{t-1} is the old state, and x_t is the input vector at time step t. This process is the general form in which a cell can learn something based on context, i.e., not only new input but also its previous output, which is fed into itself again. Figure 2.16 provides a simple illustration of a single cell within a recurrent neural network:

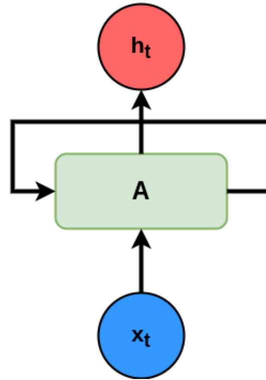

Figure 2.16: The basic structure of a cell in a RNN

RNN's are powerful and versatile in an application point of view. They can solve different types of problems by using different types of architecture. In figure 2.17, we provide an overview of the different applications available, with their respective architecture. Each rectangle is a vector, and each arrow represents functions (e.g., matrix multiplication). The red arrows represent input vectors, the green arrows hold the cell states, and the output vectors are blue.

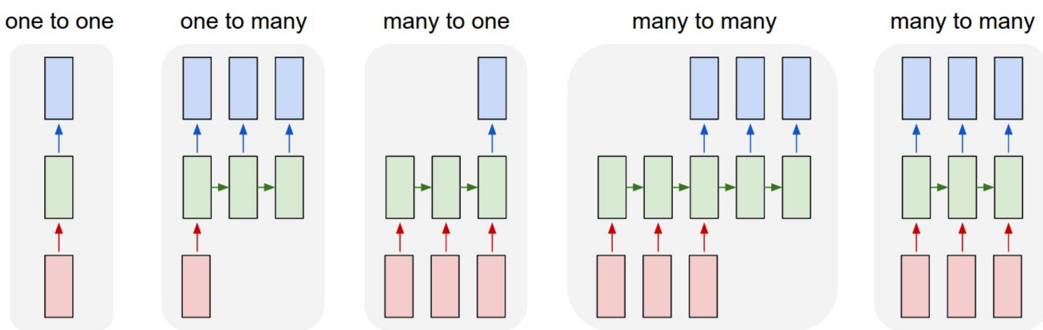

Figure 2.17: Example RNN structures and their application. [35]

One-to-one: Normal feed-forward network, without RNN, fixed-size input, and fixed-size output. Used in for instance image classification.
One-to-many: network takes an input and produces a series of outputs, for example, an image captioning problem, where the input is a single image, and the network produces a sentence of words with context.

Many-to-one: Network takes a series of inputs and produces a single output. An example could be sentiment analysis, where a sentence is either classified as positive or negative. Another example could be video classification, where the network is fed frame by frame. The RNN would then capture the order between them for a better classification.

Many-to-many (encoder-decoder): Sequence to sequence learning, for example, Machine Translation, where the network translates a sentence from one language to another.

Many-to-many: Also a sequence to sequence learning architecture, but mostly used for video classification on a frame by frame level, i.e., classify every single frame in a video [36].

Before we go into why we chose RNN for our problem, let us look more into detail on how RNN's work. From a top-down perspective, the network accepts some input vector x and outputs a vector y. However, as mentioned above, the crucial part of why RNN's are so effective is that they are not influenced only by new input vectors from our current timestep, but also the entire history of inputs provided earlier. For every timestep, the RNN has some internal hidden state h_t that gets updated. In a Vanilla RNN the formula for the forward pass looks like this:

$$\begin{aligned} h_t &= tanh(W_{hh}h_{t-1} + W_{hx}x_t + b) \\ y_t &= W_{hy}h_t \end{aligned} \tag{2.14}$$

As we can see, the vanilla RNN's trainable parameters are the three weight matrices: W_{hh}, W_{hx}, W_{hy}. Notice how the hidden state is calculated with a non-linearity $tanh$ function that squashes our activations between [-1, 1]. Notice also that the hidden state is dependant on the previous hidden state h at timestep $t-1$ and the new fresh input x at the current timestep t. In practice, most do not use the vanilla RNN model but rather a more advanced model called the Long Short Term Memory (LSTM) network and the Gated

Recurrent Unit (GRU). Both of which we will touch briefly upon. LSTM and GRU have proven to work better in practice due to a more powerful update dynamic.

2.4.1 Long Short Term Memory (LSTM)

To understand how the LSTM updates its state, let us take a look at a LSTM cell diagram.

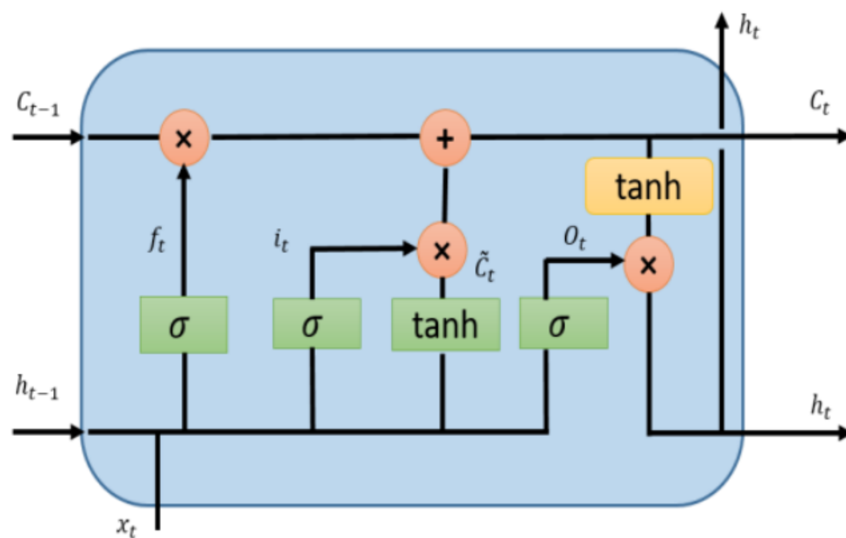

Figure 2.18: Diagram for Long Short Term Memory (LSTM). [37]

The LSTM's remove or add information to the cells state through different gates: input gate (i_t), forget gate (f_t) and output gate (o_t). The output from these gates are calculated with different non-linearity's and equations and they are defined as:

$$
\begin{aligned}
&\textbf{Input gate:} \quad i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
&\textbf{Forget gate:} \quad f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
&\textbf{Output gate:} \quad \tilde{C_t} = tanh(W_C \cdot [h_{C-1}, x_t] + b_C) \\
&\textbf{Potential cell memory:} \quad C_t = f_c * C_{t-1} + i_t * \tilde{C_t} \\
&\textbf{Output gate:} \quad O_t = \sigma(W_O \cdot [h_{t-1}, x_t] + b_O) \\
&\textbf{Final cell state:} \quad h_t = O_t * tanh(C_t)
\end{aligned}
\tag{2.15}
$$

The different inputs and outputs to the LSTM cell are:

$$
\begin{aligned}
&\textbf{New cell state:} \quad C_t \\
&\textbf{Cell state / Long-term memory:} \quad C_{t-1} \\
&\textbf{Output / New hidden state:} \quad h_t \\
&\textbf{Hidden state / Short-term memory:} \quad h_{t-1} \\
&\textbf{Input:} \quad x_t
\end{aligned}
\tag{2.16}
$$

To bring some further insight into the inner workings of an LSTM cell, we would like to present a brief explanation of the objective of each gate.

Input gate: The input gate decides what new information to store in the long term memory. From the equations, we can observe that it only operates on the current input and short term memory from the previous step. This gate works like a filter, discarding inept information, and keeping essential values. The *sigmoid* function will transform the input values between 0 and 1, where the decimal range decides whether or not the information is useful. 0 indicates that information is unimportant, and 1 indicates that the information is useful. From the figure, we can also observe that input is fed through a second activation function, which is used to regulate the network. Both outputs from *sigmoid* and *tanh* are multiplied together.

Forget gate: The forget gate decides which information from the long-term memory should be kept or discarded. A decision is made by multiplying

the incoming long-term memory by a forget vector generated by the current input and incoming short-term memory. The forget gate also works like a filter for which information to keep/discard. Outputs from the Input gate and the Forget gate will undergo a pointwise addition to calculate a new long-term memory. This new long-term memory is applied in the final gate, the Output gate.

Output gate: The output gate is responsible for calculating and creating the new short-term memory and the hidden state, which are passed to the cell in the next step. These values are calculated by using the current input, previous short-term memory, and the new long-term memory values. Previous short-term memory and input will be passed through a *sigmoid* function and thus acts as a third filter. The new long-term memory is passed through a *tanh* function before both outputs are multiplied together.

The short-term and long-term memory produced by these gates is carried over to the next cell, and the process will repeat.

2.4.2 Gated Recurrent Unit (GRU)

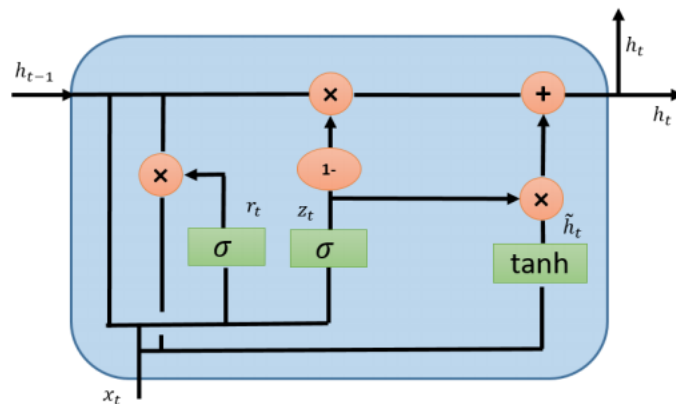

Figure 2.19: Diagram for Gated Recurrent Unit (GRU). [37]

GRU is also a more advanced recurrent unit, and like LSTM it uses gates to control information flow. As with LSTM, GRU's are used to improve the long term dependencies of information. Both of these models have the ability to add and remove to the state, not just transforming it. In GRU, the gates and cells are defined as:

$$
\begin{aligned}
&\textbf{Update gate:} \quad z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \\
&\textbf{Reset gate:} \quad r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \\
&\textbf{Candidate cell:} \quad \tilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t]) \\
&\textbf{Final cell state:} \quad h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
\tag{2.17}
$$

There is no conclusive evidence of whether LSTM or GRU is better for learning sequences. However, GRU requires fewer network parameters, which makes training and convergence faster. On the other hand, LSTM provides better performance, so with enough data and computational power, it is often preferred. [38]

2.4.3 Why we choose RNN's

The main reason why RNN is our preferred deep learning strategy is because of their powerful ability to capture context through data and to process input vectors of different lengths. We are absolutely dependant on the flexibility that RNNs can provide us. Especially when we considerer training data of different variations and lengths. There are many ways to write programs containing buffer overflow weaknesses. Our neural network must, therefore, be able to discover a pattern within all these different ways of writing code and learn from them. The very same holds for the length of our training data as well. Other deep learning architectures such as CNNs and vanilla feed-forward networks are dependant on uniform length input vectors.

2.5 Summary

In this chapter, we have presented the theoretical underpinnings we believe required to appreciate our dissertation. We have given a thorough elaboration on stack-based buffer overflows, their history and relevance, and provided a small example of how this vulnerability works. We continued with a bottom-up perspective on neural networks where we built up a good intuition of its inner workings. We started with the atomic parts, which make up a neural network and gradually pieced everything together to compose a stack of neurons, weights, and layers. We presented the fundamental principles behind machine learning, which included: activation and loss functions, backpropagation, gradient descent, and regularization. In the following subsection, we provided an understanding of the main categories of deep learning, how they differ, and their objectives. Finally, we presented the inner working of a recurrent neural network. We focused on the inner workings of recurrent cells, their capabilities, and why we choose to use them in our experiments.

Chapter 3

Literature review and taxonomy

This chapter function as a summary of current and previous work that has been done in our field of research. As of today only a marginal amount of study on the possibilities of vulnerability detection through deep learning has been done. We are to present our inspected papers in a categorical way in which we sort papers into subsets based on their features and architecture.

We believe our literature survey is beneficial in multiple ways. First and foremost, we can take a look at how researchers have chosen to represent their data, what kind of deep learning architecture they have used, and what kind of different techniques (activation functions, regularization, optimization) have been proven effective. From the few research papers we have found that explore the same problem as us, we can get an indication of their success, thus giving us some insight into possible approaches.

Much of the research, as we will soon see, has been done by teams with a vast amount of resources. Some of the more successful work has been collaborations between multiple scientists who had access to significant computational power and data. We do not have those kinds of resources available, and since both computational power and data are the two most important factors of deep learning, we are somewhat limited. Below we have presented some relevant research papers. We have included a summary of their work, data representation level, the architecture used, and results achieved.

In this literature review we differentiate between **dynamic** and **static** features. In this context, static features are extracted without executing a data sample, i.e., program file. Dynamic features require the execution of the program. We have chosen to split the relevant research into four main categories: **Static features with standard ML approaches**, **Static features with CNN**, **Static features with RNN** and **Dynamic features with standard ML approaches**. This way, we can present the different approaches within the field of research in a straightforward way.

Paper	**RL**	**Model**	**Datasize**	**Features**	**Objective**
EMBR[39]	Extracted features, PE headers	Gradient boosted tree	>1,100,000 samples	Static	Malware detection

TDBP[40]	PE imports, headers, byte entropy & strings	FCC	>430,000 samples	Static	Malware detection
BTDA[41]	Binary code transformed into greyscale images	k-NN	>100,000 samples	Static	Malware classification
MIAC[42]	Binary code transformed into greyscale images	k-NN	>9,000 samples	Static	Malware classification
DMME[43]	Byte sequence, DLL's, function calls	RIPPER, Naive Bays, MC system	$\sim$ 5,000 samples	Static	Malware classification
VPTA[44]	Java source code	SMV	$\sim$ 2,300 samples	Static	Vulnerability prediction
PVFE[45]	Java source code	SMV	-	Static	Vulnerability detection

VEML[46]	C source code	PCA	> 6,700 functions	Static	Vulner-ability detection
MDEE[47]	Byte string	CNN/FCL	>2,000,000 samples	Static	Malware detection
MDMD[48]	Extracted features, PE headers	FCC, LSTM, Extra tree, Random Forest, LR-3 gram	240,000 samples	Static	Malware detection
VDDR[49]	C source code	Comb. of BOW, CNN and RNN	> 12,000,000 samples	Static	Vulner-ability detection
VDP[50]	C source code	BLSTM	>60,000 samples	Static	Vulner-ability detection
CAD[51]	Network Packets	RNN LSTM	< 5,000,000 samples	Static	Intrusion detection
DLCS[52]	Kernel sys calls	FCC, CNN and CNN+LSTM	>430,000 samples	Dynamic	Malware classifica-tion
NGMD[53]	N-gram	k-NN	2,000 samples	Dynamic	Malware detection

VDDL[54]	Kernel sys calls	CNN, LSTM, CNN+ LSTM	> 9,000 samples	Dynamic	Vulner- ability detection
VDML[55]	C / ASM source code	Linear re- gression, MLP, Random forest	>138,000 samples	Static/ Dynamic	Vulner- ability detection

***RL:** Representational level

3.1 Static features / standard ML approaches

We would like to start with a section about the standard machine learning approaches on static features. With standard machine learning approaches, we include standard feedforward deep neural networks and any machine learning techniques such as gradient boosted trees, K-NN (k-nearest neighbor algorithm), and support vector machines (SVM). Refer to [56] for further enlightenment.

We would like to start our literature review with a project which perhaps can be said to be the gold standard within the field of research: the EMBER [39] project. This paper describes the use of a gradient boosted decision tree to detect malicious Windows portable executable files statically. The EMBER dataset consists of features extracted from 1.1 Million binary files with 900,000 training samples (300,000 malicious, 300,000 benign and 300,000 unlabeled) and 200,000 test samples (100,000 malicious, 100,000 benign). The samples are structured as JSON objects containing selected features from the Windows Portable Executable (PE) file format [57], which were hashed for dimensional reduction. Some features included general file information, header

information, imported functions, exported functions, and format-agnostic histograms of string occurrences. The EMBER dataset is, in fact, the first large dataset for machine learning malware detection. Results showed that even without hyperparameter optimization, the baseline EMBER model outperformed the highly successful MALCONV [47].

One of the only successful uses of a deep, fully connected neural network for malware detection was the one by Saxe and Berlin. In their paper: Deep neural network-based malware detection using two-dimensional binary program features [40], they explored malware identification/classification through standard deep learning tools. Their network was a basic four-layer deep feedforward neural network with Parametric Rectified Linear Unit (PReLU) activation functions in the first two hidden layers. Their benign and malicious malware were drawn from Invincea's [58] own computer systems. The final dataset consisted of a total of 431,926 binaries (81,910 benign and 350,016 malicious). The feature engineering consisted of a Byte/Entropy histogram, hashed PE import address tables, DLL names, and import functions. These feature vectors were concatenated into a 1024-dimensional vector, which was the input to their neural network. Saxe and Berlin also tried to train the network on a subset of their features to get some insight into which features that contributed the most (at least by themselves) to the classifier. Impressive results were achieved, and the paper reported the highest accuracy by using all features and the highest contributing feature being metadata from the PE headers.

Moving over to k-NN's, the first paper we would like to present is a comparative assessment of malware classification [41]. In this paper, researchers did not only build a model capable of classifying binaries, but also made a comparison against classical dynamic classification methods. Their chosen methodology was to create images or image-like textures from malicious and benign malware and then to utilize classical image classification techniques. Their dataset consisted of about 100,000 malware examples divided into six categories of malware. Nataraj et al. created their features by transforming each binary malware into a grayscale image and concatenated these images

with other features. They compared their k-NN classifier against two state-of-the-art dynamical dataset analyzers: Host-RX and Malheur. The average accuracies for the k-NN classifier and the dynamic analysis tools were excellent, and they both achieved relatively identical results. Even though the accuracies were relatively similar, the amount of time required to complete the feature analysis per binary is approximately 1/4000 of that compared to the dynamic analysis tools. These are promising results concerning the usage of deep learning tools as classifiers on binary executables.

Another relevant paper on malware classification using the k-NN algorithm is [42]. Exploration of effective methods for classifying malware through grayscale images created directly from the binaries was conducted. Malware binaries were read as a vector of 8-bit unsigned integers and then organized into a 2D array (grayscale image) in the range [0-255]. Texture features were created by using GIST. During development, they experimented with a small scale dataset consisting of 1,713 malware images split between 8 different malware families. Their large scale experiment was done on a larger dataset of 9,458 samples with 25 different malware families. Results were promising, reaching an exceptionally high accuracy. Interestingly, even obfuscating the malware by packing it (which is common to avoid detection) did not lower the accuracy of their classifier. These results are auspicious, as they prove that machine learning models can achieve not only high accuracy but also resilience against obfuscated malware.

The last paper on malware classification by using standard machine learning approaches on static features is about using data mining for malware detection [43]. Researches explored the possibility of using data mining methods as classifiers for malware. Their dataset consisted of 4,266 programs split into 3,265 malicious binaries and 1,001 clean programs. All programs were mined from public sources. Features used for training were extracted from each program and included byte sequences (using hex dump), DLL used, DLL function calls, and the number of times such functions were called. Researchers explored four different learning algorithms: a signature-based classifier (as a benchmark), an inductive rule-based learner (RIPPER) [58], a probabilis-

tic method, and a multi-classifier system that combines the outputs from several classifiers (voting) to generate a prediction. The signature-based approach (byte sequences), was the most straightforward implementation and achieved the lowest results. Their rule-based algorithm had a much higher detection rate, but the highest false-positive rate. Both the probabilistic and multi-classifier used the Naive Bayes algorithm, where the multi-classifier used multiple instances of Naive Bayes. Both approaches had an excellent detection rate and overall accuracy. Surprisingly their single Naive Bayes achieved the lowest false positive rate.

Proceeding to vulnerability detection and prediction, which is the most relevant work for this thesis, we would like to introduce a paper that used Java source code at their representational level. [44] Hovsepyan et al. explored the ability to do binary classification, i.e., vulnerable, not vulnerable, and the severity of said vulnerabilities on Java files. Their approach is quite fascinating, as they treat Java files as pure text files. The data used was source code from 19 different versions of the K9 mail application [59], adding up to a total of about 2,300 Java files. Samples were labeled by using the Fortify [60] vulnerability analysis tool. Features were generated by tokenization over each binary, thus creating a vector representing the occurrence of each textual "word" in a binary. The way they created feature vectors was by treating every word in a binary as a feature. The feature vectors were fed into a SVM for both the training phase where the prediction model is built and the prediction phase where new feature vectors are classified based on the previously built prediction model. On average, the SVM model achieved good accuracy, precision, and recall.

Another paper that used Java source code, as well as a SVM classifier, is a paper by Pang, Xu, and Namin [45]. They proposed a hybrid technique based on combining n-gram analysis and features selection algorithms for predicting vulnerable software components in some commonly and widely used applications written in Java. Subject programs were "BoardGameGeek", "Connectbot", "CoolReader", and "AnkiDroid". N-grams were created by analyzing and building a vector that would hold the frequency of appearances

of a given token. These features were directly derived from Java source code, so no execution of the binaries was necessary. Labels for their dataset were available from previous research done on the same applications. Feature selection methods were based on the Wilcoxon test [61] for reducing space and dimensions of features. Their actual learner, or model, was elected by comparing the performance of six different known classification algorithms. Best results were achieved with support vector machines, and therefore was the chosen focus on their larger dataset. The proposed technique could classify vulnerable classes with high precision, accuracy, and recall.

Concluding the static features on standard machine learning techniques, we present a paper that uses C source code as representational level. [46] is a paper where researchers proved excellent capabilities and potential of assisted discovery of vulnerabilities by using a known machine learning technique known as Principal Component Analysis (PCA). Yamaguchi et al. used a known vulnerable library called FFmpeg, a library for multimedia processing. Using a known vulnerability as a label, they performed a mapping of the rest of the source code, which has a total of 6,778 functions. Mapping was done through four steps: Extracting names of types, names of functions, and typecasts referred to as API calls. Next, they performed an embedding on API calls to create a vector, where each dimension represents one API symbol. Before performing the actual vulnerability discovery, they also used a PCA to enable the model to infer descriptive directions in the vector space. Their research discovered that their model could detect not only a previously patched vulnerability but also discovered a new function vulnerable to exploitation. This finding is indeed interesting, as it proves the power and capability standard machine learning techniques can have in assisted vulnerability discovery.

3.2 Static features, convolutional networks

Proceeding with static features, we present the only paper which heavily relies on CNN's and static features. Paper [47] is also one of the gold standards

of malware detection. The goal for the Laboratory for Physical Sciences and NVIDIA was to do malware detection from raw byte sequences. They performed a static analysis approach, where a higher-level representation was constructed from the raw byte inputs. The datasets used for training consisted of 2,011,786 binaries split evenly between malicious and benign classes. Group B contained binaries ranging from 1MB to 2MB in size. An anti-virus industry partner provided these binaries. Group A was collected through data mining for finding new malicious executables and "goodware" from a clean Microsoft installation. The chosen architecture was tokenized raw bytes fed into an embedding before Convolution and finishing with a fully connected layer and softmax. They chose a very shallow architecture compared to other deep learning architectures. In conclusion, MALCONV achieves consistent generalization across both test sets, despite the challenges of learning a sequence problem of unprecedented length.

3.3 Static features RNN

The last sub-category of research which utilizes static features are pure, or partly recurrent network architectures. These are the most relevant to us as we opt to explore the same architecture, albeit different representational levels. Some papers present CNN's as well, but rather as a benchmark or experimental architectures in a set of different deep learning approaches. Again, we will start with malware detection. First is a comprehensive approach with several architectures. The overall focus of [48] was to explore the possibility for neural networks to learn its feature representation from raw data. Raw data in this context was bytes extracted from PE headers as with paper [39]. The general goal was to train a model that could identify and differentiate between malicious and benign executables. Each input vector consisted of a 328-byte representation of selected features extracted with the intent to minimize preprocessing. These features included pointers to the Import table and various other information extracted from the PE headers. Training and testing data were created through a combination of datasets provided

by Virus Share, Open Malware, and MS Windows. From our perspective, this paper is very appealing as Raff et al. explored and compared results from five approaches: Fully connected, LSTM, Extra Tree(Extremely Randomized Tree), Random Forest, and LR 3-grams (which is a form of n-gram). Interestingly enough, the most successful architecture reported was the fully connected architecture. The other four architectures performed reasonably well.

Next up are two papers which are very relevant to us. They use some sort of RNN, static features, and does vulnerability detection. The first paper [49] also holds the most considerable sample quantity we have seen so far. Russell et al. present a machine learning technique for the automated detection of vulnerabilities in C / C++ source code learned from real-world examples. In one way, their approach is different from what we try to achieve, as we simulate the absence of such source code. They created their data by compiling a vast set dataset of millions of function-level examples of C and C++ code from the SATE IV Juliet Test Suite, Debian Linux distribution, and public repositories on GitHub. Their total number of functions included in the dataset exceeded 12 million samples. Input source code was first filtered through a series of convolutions before being fed into a sequence of RNN's. These steps created their learned source features, which were finally fed through a random forest classifier and fully connected classification layers. They also trained a RF classifier on a Bag-Of-Words (BOW) representation as a benchmark. An additional comparison was made between three static analyzers and their own implemented architectures. All their architectures performed very well and outperformed the static analyzers by a large margin. The most successful architecture was the RNN + CNN, which performed outstandingly on both datasets.

Some highly interesting research has been done by using a unique form of RNN's, the bi-directional LSTM. The VulDeePecker [50] is a model developed for vulnerability detection. The team of researchers derived features directly from C source code. These features were created by extracting library and function calls. Through these features, they generated assembled

slices of code, which they called code "gadgets". These gadgets are semantically related lines of code that were fed through several BLSTM layers, a fully connected layer, and finally, a softmax layer. Gadgets were generated from a set of 61,638 code samples provided by the National Institute of Standards and Technology (NIST)[62] and the Software Assurance Reference Dataset (SARD)[63] project. The paper reports excellent success, with low false-positive rates and high overall accuracy. These results were compared against three other pattern-based vulnerability detection systems: Flawfinder, RATS, and Checkmarx. All of which yielded inferior results compared to the VulDeePecker.

Some work has also been done with intrusion detection using RNN and static features. Even though the work performed in this paper is not as related to our task in this dissertation, the architecture and techniques used are valuable to consider. Cao et al. [51] created a model for intrusion and anomaly detection using deep learning techniques similar to the ones we opt to implement. More specifically they utilized a RNN with LSTM cells to detect and classify **neptune** DOS attacks[64]. They trained and tested on the KDD 1999 dataset [65], a CSV file of time series, containing a set of standard data which includes a wide variety of intrusions simulated in a military network environment. The KDD 1999 consists of almost 5 million entries where each sample is a feature vector of packet frequencies derived from a **tcpdump** of network activity. Their experiments yielded promising results as the model was able to capture the anomalously high number of SYN_ACK packets associated with the Neptune DOS attack. It is indeed interesting that RNN LSTM cells were able to capture this context and differentiate between Neptune DOS attacks, other intrusions, and normal network traffic.

3.4 Dynamic features standard ML approaches

Even though dynamic features are quite the opposite of our static features, there exists much intriguing research for malware/vulnerability detection and classification. A quite exciting approach by Kolosnajaji et al. [52] are models

training on dynamic features extracted from kernel API call sequences. The system calls were transformed into a unique binary vector for every API call present in the dataset. The dynamic features were extracted by feeding each data sample through a dynamic malware analysis tool to gather the underlying data about malware behavior. Kolosnjaji et al. compared three types of architectures: a feedforward network, a convolutional network, and a hybrid neural network made up of a convolution part and a recurrent part. In the hybrid implementation, outputs from the convolutional part are fed into a recurrent part consisting of LSTM cells. All the binaries used for training and testing are gathered from Virus Share, Maltrieve, and private collections. They created 10 clusters of classes where each cluster consisted of 4,753 data samples. The researchers were able to achieve good results where the hybrid ConvNet + LSTM yielded the highest average performance.

Another paper by Santos et al. [53] explored the possibility of using a classic language recognition technique called n-grams for malware detection. N-grams are substrings of a larger string of a fixed length n. In total, Santos et al. used a subset of 1,000 benign and 1,000 malicious software provided to them by a computer security software company. For classification, they used a k-NN algorithm on n-grams created from file signatures of a collection of malware and benign files. The signatures were created dynamically by running each binary in a safe environment and logging each binary's behavior. Researchers tuned the length/size of different n-grams and reported that they achieved the highest accuracy by setting n-grams to 4, which yielded a high detection ratio.

Wu et al. [54] explored the possibilities of vulnerability prediction based on a dynamic analysis of software. They trained three deep learning models: a convolution neural network, a long short term memory recurrent neural network, and a hybrid of the two. They also implemented a very basic Multi-Layer Perceptron (MLP) for benchmarking the other architectures. Their dataset consisted of a collection of 9,872 sequences of kernel function calls. Functions were pulled from the **/src/bin** and **/usr/bin** directory in a 32-bit Linux machine. For the CNN architecture, input was a sentence

comprised of word embeddings. For the LSTM model, the first layer is an embedding layer that uses ten length vectors to represent each word. For the hybrid model, input in first fed through a CNN part before it is available for the LSTM cells. On average, the three models achieved fairly reasonable results compared to the vanilla MLP model. Once again, the CNN+LSTM was the most successful architecture, albeit with some small margin.

We conclude this literature review with a paper that uses a riveting representational level. In this paper, through the use of static and dynamic features, researchers presented a way to predict if executables contained vulnerable memory corruptions by using machine learning techniques. They collected 138,308 unique execution traces and statically explored 76,083 different subsequences of function calls. Static features were based on the use patterns of the C standard library within a binary. Static feature vectors were made by doing a random walk on parts of the executable and producing feasible/unfeasible sequences of C library calls. Dynamic features were extracted by hooking program events and collecting them into a separate sequence. Grieco et al. [55] explored three different machine learning techniques: a logistic regression model, a Multi-Layer Perceptron (MLP), and a random forest, where the latter gave them the highest prediction accuracy. Their random forest model managed to predict with reasonable accuracy which programs contained dangerous memory corruptions. Even though their models' accuracy was not the most impressive, it still holds as a supporting tool for vulnerability discovery.

3.5 Summary

In this chapter we have given a thorough presentation of relevant work in our field of research. Each paper has been briefly presented with their approach, data, representational level and findings summarized. We have established a understanding of recent and current achievements in malware detection, malware classification and vulnerability detection using machine learning techniques. A taxonomy of our chosen papers was created to provide a quick overview of representational level, models used, size of their datasets, features used and overall objective.

Chapter 4

Challenges and Limitations

This chapter will provide insight into an understanding of our limitations. Our work is only part of a master thesis; thus, our aim is, as expressed in the motivation section, only to find and draw basic conclusions on whether or not our approach is meaningful and useful. We do not, in any way, claim that our work generalizes over other machine learning inspired approaches. Non-identical data sources, feature engineering, architecture, or neural network classes could yield entirely different results and conclusions. In the context of our research method, implementation and limitations, we strive to shed light on and evaluate the feasibility of our particular approach.

4.1 Computational limitations

We have been quite limited concerning computational power and the availability of said resources. At the time of writing, the Abel [66] cluster at UiO was in the phase of permanently shutting down. We are indeed appreciative of UNINETT, providing us with high-performance computing and data storage. Our resources were relatively limited. Even though our resources were limited, as they are intended for small scale projects [67], we would not be able to conduct our experiments without them. The lifespan of a SSEW project with HPC is limited to three months, which is unfortunate. It has limited our ability to conduct as many experiments and explorations as we would have wanted. Given the size of each sample, loading and training may take a long time. We have therefore been obliged to be relatively conservative when deciding the number of functions in each sample and the total number of samples within a dataset. These computational restrictions have influenced everything from data gathering, generation, feature engineering, architecture, and experiments.

4.2 Data limitations

A measure for reducing and limiting our scope is to concentrate on a single type of vulnerability, buffer overflows. For us, it was a natural choice, as our research goal is to conclude our approach's usability. Thus we cannot draw a generalized conclusion for our model's capacity to classify other vulnerabilities as their code signature might differ severely. This limitation is directly related to our available resources, both human and computational alike. As we generate, label, and prepossessing all data by ourselves, it is essential for us to keep the complexity of our work at a manageable degree. Even though we are restricting ourselves to a single type of vulnerability, we would still argue that this approach gives us a sufficient basis for making conclusions. Further discussions and settlements about our datasets are found in the approach and implementation chapter.

4.3 Classification limitations

The last category of limitations is the classification task itself. We opt for a model that would be able to detect whether or not a binary has **a** vulnerability or not. Which in this context means that we do not concern ourselves with either counting the number of vulnerabilities a binary has or its position within the binary. The ability to detect multiple vulnerabilities and pinpoint their position would be an incredibly complex task, which we neither have time or resources to implement. Even though this certainly would be an exciting extension of our work, it is not imperative for our hypothesis.

4.4 Summary

In this short chapter, we presented our limitations and challenges and thus established our scope of research. Albeit our limitations and said scope, we argue that our findings still hold merit within the limitations and constraints defined above.

Chapter 5

Approach and Implementation

In this chapter, we aim to provide an overview of our approach for creating the necessary segments to conduct our experiments. This overview includes detailed steps of how we generate data, how we create features through data cleaning, and, finally, an overview of our architecture. We have included multiple listings and figures to ensure a fundamental understanding of our research method and implementation.

5.1 Data Generation

As with any deep learning problem, the quality and amount of data available for training, validation, and testing are critical to the success of a model. One of the major and perhaps the most significant challenge of our problem is the lack of such data. As of today, to our knowledge, there does not exist a database or data collection with the data needed for our purpose. The lack of such a database is a multifold challenge. First of all, we had to figure out a way to gather or create this type of data. Secondly, we had to decide how we want to represent our data in a meaningful, compact way while keeping the representation as close to runnable code as possible. Lastly, we also need to categorize/label each dataset so that our model can learn from training data and categorize/label unseen samples in the future. Let us first examine the possibilities and limitations of different approaches to creating a suitable dataset for our model.

5.1.1 Web scraping

The initial idea was to use a web scraper to scrape different C source code sources, which is considered a reasonable solution in general. Sources to consider would be public repositories at Github, open-source code for large software projects, or even code from different open-source operating system kernels. The positive aspects of this approach are that the code gathered will have significant variance and be non-homogeneous as developers write code in different styles and with different conventions. Strictly speaking, non-homogeneity might be a challenge, but most deep neural networks can deal with this type of difference quite well. Some of the papers we have explored utilized this approach to create a large dataset with a massive collection of samples. These papers were concerned with malware detection.

Another great benefit of gathering data is that this approach will almost completely negate any unintentional tailoring or bias compared to writing/-generating the data. Another argument for scraping the web for data is that the collected data is arguably more "real world" like, and would perhaps

converge the model towards a more "universal" applicable state. In turn, the model could be better at classifying unseen examples from very different sources or code written in unique ways. In general, it is relatively safe to say that this approach would yield high-quality data samples and, in turn, high-quality datasets for training and testing the neural network. Even though there are great benefits to gathering data samples from web/repo/project scraping, there are some challenges to overcome. First of all, we would have to create a web-crawler / web-scraper that would have to navigate through the web, find C code, extract it in a usable, and compilable format (which can be quite hard for nested calls), and write it to a file.

A significant concern from our part is that there is no guarantee that the collected data would be of high quality. Indeed there exists a wast amount of unmaintained code on the web. There is no way for us to know for sure that code gathered would even compile without us testing each gathered sample. Quality checking each data sample adds a great deal of work, which can be avoided using other approaches, which we will present below. Crawling and extracting data with this approach is quite resource-heavy, but doable. The greatest and perhaps the most limiting challenge is labeling the gathered data. For every single data sample, we would have to perform an in-depth analysis to determine whether or not it contains a buffer overflow vulnerability. Since we are using supervised learning, this is necessary if we want our data to have any value for our model. There are, in general, two ways we could go about labeling the scraped functions; manual or with the help of automatic tools.

A manual approach to labeling

A manual approach of labeling would require months and months of tedious analysis of source code by a dedicated team of reverse engineers as the amount of data required for deep learning is vast. Manual labeling of data has been performed with great success in other datasets, for example, the CIFAR-10 and CIFAR-100 [68] datasets used in image classification. Students manually labeled both these datasets at Toronto university. CIFAR-10 has 60,000

image samples over ten different classes, so it is quite large, but not nearly as large as many other datasets used for deep learning and image classification. We would argue that compared to image labeling, vulnerability labeling is a whole different ballpark. Looking at a picture and determining what kind of animal it is, is relatively trivial for most human beings. Looking at source code and determining whether or not it contains a particular vulnerability is quite hard for most of us. Some professional reverse engineers have more training in this work, and would thus be faster, but obscure examples can fool even the professionals. The only way to have 100% correct labeling would be to run the code with different inputs and try to crash or cause a buffer overflow through each function by writing past allocated memory to smash the stack. This is an enormous task, and the resources required for this manual work are not available for our project. In general, it is not a feasible or even time-effective approach when labeling such complex data.

An automatic approach of labeling

The second way for labeling all collected functions would be to use some automated vulnerability detection tools. Using detection tools would be much faster and less resource consuming than the manual way. The challenge with automated tools is that they, as humans, are prone to error. So even though a vulnerability detector is a great tool, it is only as good as the engineers who programmed it. They are prone to do wrong labeling, which would be detrimental to our deep learning models. A possible solution for this inaccuracy would be to do a sort of voting on labels. So if we were to run each data sample through a set of automatic tools, we could decide the label based on a majority vote or a certain percentage. The best approach would probably be to use a set of both static and dynamic analyzers, as both categories have their strength and weakness. Even though this automatic approach is much faster than manual labeling, it still takes time for each detector tool to run all its fuzzing and different inputs for each function/data sample. We would also argue that a neural network trained on data labeled by a set of automatic vulnerability detection tools may just end up learning the function encoded

by the automatic tools. If this scenario emerges, it would be quite useless for us since there would be no point in training a resource expensive classifier of a collection when automatic tools would already provide the desired function just as well.

5.1.2 Generating & creating data

Our approach to the lack of finalized datasets for this type of deep learning task is to create our dataset. In general, generated data is not considered quality data in deep learning. However, with code, we would argue that if the model can infer context over our generated examples, so should the model do with code created by many different software engineers. A great benefit of generating our data is that labeling is straightforward. Since we are creating each function, we know exactly if a sample is vulnerable or not. We also have full control over the code style used, so our data is more homogeneous, which can be beneficial for our neural network to converge faster.

Our approach of training our model with synthetic data is a contribution towards the feasibility of vulnerability detection using Recurrent Neural Networks. Synthetic data is thus, in our view, sufficient quality to prove/disprove our highly non-trivial hypothesis. If we can prove the feasibility grounded in synthetic data, extending the model to work with more complex and realistic data is mainly an engineering challenge, which again is highly non-trivial. On the other hand, one of the significant drawbacks is the amount of data and how varied each sample can be when we choose to write it ourselves. These drawbacks are mostly due to time constraints, as one person cannot sit down and write code samples in the 100,000's size. We chose a different and interesting approach in which we mitigate these drawbacks.

Let us start by defining what a vulnerable function is for our purpose. For us, the only vulnerability we focus on is the buffer overflow weakness; this means that we have only focus on and create functions that are susceptible to that particular vulnerability. We have also decided that our vulnerable functions must take some input from the user. Below is an example of such

a function, and a function which we do not deem vulnerable in our context. In this example **strcpy** is used.

```
void notSafeCopy(char* src){
    char dest[14];
    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
}
```

Listing 5.1: Buffer overflow vulnerable data sample

Listing 5.1 presents a vulnerable function, where argument src larger than 14 bytes will overwrite allocated dest memory.

```
void notSafeCopyNoArg(){
    char* src = "This is a string that is too long for
    dest"
    char dest[14];
    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
}
```

Listing 5.2: A crashing function

Listing 5.2 provides a function where, if run, will crash the program that calls it. We would argue that this function only has a programming error that will crash the program but is not exploitable by malevolent users. Buffer overflow vulnerabilities can be written in numerous ways, but in general, they are produced through unsafe native C system calls. That is, without doing appropriate buffer length checks. We choose to focus on system calls that are considered unsafe and use system calls that are considered safe and perform buffer length checks. Even though the last category is considered safe, there are ways for the programmer to make mistakes, which will open up vulnerabilities nonetheless. Here is a list of our chosen system calls by category, with a short description of its usage, and why it is unsafe:

Copy a string :

strcpy: char *strcpy(char *dest, const char *src); Copies the string pointed to by src, including the terminating null byte ('\0'), to the

buffer pointed to by dest. If the destination is not large enough for the src string, we have a buffer overflow vulnerability [69].

strncpy: char *strncpy(char *dest, const char *src, size_t n); Works the same way as strcpy but additionally, you can limit the number of bytes copied through size_t n. However, if size and src are greater than the bytes allocated for dest, we have a buffer overflow vulnerability [70].

Concatenate a string :

strcat: char *strcat(char *dest, const char *src); Appends the src string to the dest string, overwriting the terminating null byte ('\0') at the end of dest, and then adds a terminating null byte. If dest is not large enough for src, we are invoking unpredictable behavior and creating a buffer overflow vulnerability [71].

Input format conversion from STDIN :

scanf: int scanf(const char *format, ...); Scans and converts input according to a specified format, and the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments. If the result after conversion is larger than the destination, there is a buffer overflow vulnerability [72].

Formatted output conversion :

sprintf: int sprintf(char *str, const char *format, ...); Is in the printf family and produce output according to a format. Sprintf assumes arbitrarily long strings, and this, if the destination buffer is too small for the input, there is an opening for a buffer overflow vulnerability [73].

Input of characters and strings :

gets: char *gets(char *s); Reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte. No check for buffer overrun is performed,

so it is vulnerable. Even though gets is no longer supported, and is advised to avoid, usage is still present, and we find it valuable to include to our vulnerable functions [74].

fgets: char *fgets(char *s, int size, FILE *stream) Reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored in the buffer. A terminating null byte (aq\0aq) is stored after the last character in the buffer. Fgets is considered the safe, updated version of gets(), but it can still be vulnerable to buffer overflow if the programmer accepts a too large amount of bytes into the destination buffer [75].

Copy memory area :

memcpy: void *memcpy(void *dest, const void *src, size_t n); Copies n bytes from memory area src to memory area dest. As with the other functions, one can specify the number of bytes to copy/read. The programmer can make an error and accept a too large amount into a too-small buffer, which opens up the buffer overflow vulnerability [76].

We are aware that there are other unsafe system calls in the C library, but we have limited ourselves to these eight. For each of these eight system calls, we created 15 different functions, which means we created 120 different vulnerable functions. These functions had one and only one vulnerability and used one and only one weak system call. Our reasoning for this is to isolate vulnerabilities to buffer overflows only and to simulate "real world" data as much as possible. We strive to have a clean data sample where the vulnerability is a single type of system call and single vulnerable function.

Some functions use system calls that are considered unsafe concerning stack smashing but are written and checked in a way that makes them safe. We wanted to do this to ensure our network would not merely learn that ALL calls to, for example, **strcpy** are wrong. It depends on the context as much as the actual system call itself. We will come back to and use the same

reasoning when we provide insight and examples of our benign functions. In 5.3, we can see an example of a function that uses an unsafe C system call and thus has a potential vulnerability. However, the input is handled in a safe and controlled manner by checking the string length and setting up a correct size buffer.

```
1  void safeCopyStrcpy(char* src){
2      char dest[14];
3      if(strlen(src) > sizeof(dest)){
4          char longer[strlen(src)];
5          strcpy(longer, src);
6          printf("Copied string: %s\n", longer);
7      }else{
8          strcpy(dest, src);
9          printf("Copied string: %s\n", dest);
10     }
11 }
```

Listing 5.3: Safe useage of strcpy

Each vulnerable function is unique in the way that they all have some variety in their code running before the vulnerable system call, some other parameter, or just some other data structure. Our functions, therefore, ensures a better variance in the data samples and helps our neural network to converge to a more generalized state, hopefully. For the benign functions, we decided to create an equal amount of that to the vulnerable functions. We also decided to use eight of our vulnerable functions and repair them, or reprogram them into safe functions. In the listings below we present a vulnerable function (Listing 5.4), and its benign, safe counterpart (Listing 5.5).

```
1 void memcpySmallIntoLarge(char* s){
2     char dest[256];
3     memcpy(dest,s,strlen(s));
4     printf("%s\n", dest);
5 }
```

Listing 5.4: A vulnerable function from our dataset

5.4 shows us how easy it is to unintentionally create a vulnerability by using the wrong parameter to the C system call.

```c
void memcpySmallIntoLarge(char* s){
    char dest[256];
    memcpy(dest,s,sizeof(dest));
    printf("%s\n", dest);
}
```

Listing 5.5: Repaired version of listing 5.4

The motivation behind this is also to challenge the network. We would like the network to convergence towards a context-based state instead of a keyword-based state. With our repaired code, we want to create data samples where vulnerable operations are performed safely.

5.1.3 Testing vulnerable & benign code

Our goal has always been to prove/disprove a hypothesis, and thus we want to keep data as simplified as possible. One of the measures already discussed is the exclusions of all other vulnerabilities other than buffer overflows. Concerning each data sample, we have chosen to keep them, in a way, as atomic and "simple" as possible. We achieve this by only including a single vulnerability per function in our vulnerability library. We also narrow the usage of system calls to a minimum per function so that each function optimally has a single vulnerability at a single point from a single vulnerable system call. This atomic structure means that testing and controlling each data sample is much easier for us. Our approach to writing and implementing each vulnerable sample was to choose a system call, write a vulnerable piece of code, and then vigorously test it before writing a new vulnerable sample. We base all samples on user input, and the size of the provided input invokes all vulnerabilities. We followed a unit testing approach to our testing, where we validate each unit of code through a range of inputs. Unit testing was done for each vulnerable function during development, before accepting it into the vulnerability library.

As mentioned, some functions contain more than one system call with motivations and reasons already mentioned previously. In these cases, testing was more challenging as we had to make sure that no additional vulnerability was introduced into the sample. On the opposite side, as mentioned, half of the benign functions are repaired samples from the vulnerability library. Both these repaired functions and the new ones we wrote has also been scrutinized and rigorously tested to make sure that they were vulnerability free. This way, it is easy for us to assign each data sample to their correct labels. Correct segregation is essential for correct labeling later on. Now that we have presented some examples of both vulnerable and benign functions, we would like to elaborate on how we use these two collections or libraries of functions to generate data.

5.1.4 Generator script

We developed our generator script in python. It is capable of combining functions from both the benign and the vulnerable library we wrote to create new fully compilable and runnable C code. It has some hardcoded string values, such as headers, the main wrapper, and brackets. In other words, code that all runnable C programs need. The finalized script will, with some stochasticity, generate the data samples we need for our problem. Here are the parameters which can be used and modified:

```
argv[1]: Number of binaries to create (1-N).
argv[2]: Defines the total amount of functions in a binary.
argv[3]: Define the total number of functions from the library to
    include in the generation.
argv[4]: Flag defining chosen class. B-benign, V-vulnerable
```

Listing 5.6: Arguments for our generator script

We decided that for the vulnerable binaries, we would only include one vulnerable function; the rest of the binary would consist of benign functions. So if we specify:

```
python generator.py 100 5 20 V
```

Listing 5.7: Sample argument stack for generating binaries

Our script will produce 100 binaries, which have five functions, where four benign functions are chosen from the 120 first functions in the benign library, and one of the five is a vulnerable function from the 20 first vulnerable functions in the vulnerability library. Concerning the number of functions per binary, we wanted to keep it within a reasonable number, not too long, so that we have a large binary and not too small so that the binary has some structure and is more like the "real world". We decided that from three and up to and including six functions per binary was a reasonable number. When it comes to the number of vulnerable functions included, we argue that this is how most vulnerable binaries are structured. Many benign and safe functions, and one that has the vulnerability, and creates all the fuzz. It is not very likely that we have a situation where a binary consists of only vulnerable functions. Another supporting argument for structuring our binaries like this is that our aim for the neural network is to do binary classification, i.e., to detect if there is a buffer overflow vulnerability. We are not concerned with how many there are or their locations. Our last compelling argument for only including a single vulnerable function is that we believe our classifier will have to be far more sensitive to vulnerabilities this way. Even though it could make training and converging challenging, it is in our best interest that our classifier can indeed distinguish the small, subtle differences existing in a vulnerability hidden in a larger program file. Now that we have covered the input arguments to our script let us examine how we create new binaries.

- All the method declarations from both the benign and the vulnerable library are stored in a list.
- All function calls from both the benign and the vulnerable library in another list.

- Full code for each function in both the benign and the vulnerable library is stored in a 2D list.

These six collections of data values are stored as strings in separate lists for later processing. All of them, including arguments for how many programs \binaries we want to create and how many functions that should be included in each program is passed to the **writeToFile()** function. To ease the reading and capture each declaration, call and function we chose to mark the benign and the vulnerably library with comment tokens:

- Before all declarations we added a comment: /***Start declaration***/ and at the end a comment /***End declaration***/.
- Before all calls in main we added a comment: /***Call start***/ and at the end a comment /***Call end***/.
- We also wrapped each function with comments: /***funcstart***/ at the start and /***funcend***/ at the end.

By using these tokens, it is relatively trivial to append declarations, calls, and actual function code to separate and ordered lists. This approach is much more comfortable than the initial idea of using regular expressions (which works poorly on code with nested curly brackets). Since the code is declared, called, and implemented in the same order, each index corresponds to the same tuple. Data is generated correctly by looping over each index in this manner.

One of the last tasks our generator script executes is changing each function name to a random string. We realized that we had to do this randomization of function names, as each vulnerable function maintains its distinct name in the binary executable. Randomizing function names is critical as we do not want our model to learn any correlation between a vulnerability and a function name. If we pass the original function names to our machine learning model, it is highly likely that the model would learn that there is a vulnerability for a given function name.

```
1 notSafeMerging:
```

```
2     endbr64
3     push rbp
4     mov rbp, rsp
```

Listing 5.8: Function name in the compiled ASM format

```c
void notSafeMerging(char* s){
    char dest[15] = "Testing second";
    printf("Before: %s", dest);
    strcpy(dest, s);
    printf("After: %s", dest);
}
```

Listing 5.9: Function name in the original C function

As we can observe in listing (5.9), the name of the function is representative of what is happening. If we keep the original function name, then it is very likely that the classifier will recognize and learn classification based on function names rather than code context. Function names will never indicate whether or not it is vulnerable in the real world, so this data is undesirable for us. Therefore, we generate a list of 12 character long randomized strings and replace each function name in the declaration, call, and actual function. Since lists are indexed, it is easy for us to tie each random string to this tuple. This process performed with the generation for each new sample, benign and vulnerable. This way, we ensure that we do not have duplicate randomized strings for both a benign and a vulnerable function.

When all the previous steps are completed, our script uses random sampling to create a list of indexes for which functions to include and write to a new compilable C program. This way, we ensure that we have a uniform distribution of functions in the program generation. When the script writes a new compilable and runnable C program, declarations are written at the top, function calls within the main function, and the full functions are finally written at the bottom of the file. We chose to let the main call each function. This structure is, of course, a simplification compared to "real world" code, as functions are often called within other functions. However, this way of structuring our code makes it much easier to generate new samples logically

and gives our data samples an immaculate and understandable structure. It is not unreasonable to say that this way of coding/structuring code is also widespread within software engineering and therefore makes our samples legitimate with regard to impersonating the "real world" code.

5.1.5 Initial ideas for choosing functions and structuring them

We tried multiple approaches for generating and combining functions into unique binaries. In this subsection, we will present some of our initial ideas and reasons for not choosing them. Lastly, we will discuss our final choice for generation based on our criteria for this master dissertation.

Cartesian product

The initial idea was actually to use the Cartesian product, where we would just specify the number of functions used in each binary. The script would then pull out the said amount of random functions and create binaries that were the Cartesian product of them. So, for example, if we specified one function, the script would be able to create 120 pure benign examples and 120 vulnerable examples. For two functions, it would take the Cartesian product between every two pairs of functions, which would result in $14,400$ pure benign, $14,400$ pure vulnerable, and $14,400$ examples. If we wanted to do three function samples, we would end up at about $1,728,000$ unique samples. The drawback of using the Cartesian product is, of course, that many functions are repeated in each sample. For instance, with three functions, we would have samples that would look like this:

```
void inputStrCat(char* s);
void inputStrCat(char* s);
void inputStrCat(char* s);
```

Listing 5.10: Uniform function pool from Cartesian product

We would argue that declaring, calling, and writing a single function multiple times is not something widespread in software engineering in general. Also,

with the Cartesian product, we would create samples that consisted of N-amount of vulnerable functions. Binaries containing multiple vulnerabilities are not something that we want in our training and testing data, as we have stated that we want our model to detect **one** single vulnerability. Besides, as previously stated, the inclusion of multiple vulnerable functions makes our samples too dissimilar compared to code we would see in the real world. With this approach, we would then have to write some logic that had to check each generated sample for repetitions. We would also need logic to remove any samples which contained multiple vulnerable function calls. On the flip side of all the limitations of this approach, we can observe that given that this logic, each function would be used and called the same amount of times, which would give us a uniform function usage distribution. This distribution is one of the goals we wanted to fulfill, as we would like our model to train on all the different benign and vulnerable functions. However, the limitations outweigh the possibilities, so we eventually scrapped this approach.

Pythons Itertools

The second idea and approach would use pythons itertool [77]. Itertool with combinations creates r-length tuples, in sorted order, with no repeats. If we look at a sample implementation below, we can see which kind of tuples itertools.combinations will create. Imagine function number 15, 30, 45 and 60 were chosen at random the following tuples are created by Itertools (5.11):

```
(15,) , (30,) , (45,) , (60,) , (15,30) , (15,45) , (15,60) ,
    (30,45) , (30,60) , (45,60) , (15, 30, 45) , (15, 30, 60) , (15,
    45, 60) , (30, 45, 60) , (15, 30, 45,60)
```

Listing 5.11: Tuples created by Itertools

With itertools, we could specify the number of functions to choose (in the example above four functions). For our four function example, we would get 15 unique tuples with no repeat of function calls. With ten functions, we would get 1023 samples, and for 20 functions, we would get $1,048,575$

samples. First of all, this would give us unique function calls, which is desirable, but on the other hand, many of the samples would have far too many function calls for the limitations we have decided. Programs with 20 function calls contain much code, and we believe that programs and data samples of such length are neither necessary nor beneficial for our deep learning model. As with the Cartesian product approach, we would have to create logic for injecting vulnerable functions inside the benign samples and vice versa. A possible solution to both these challenges would be to use a subset of the tuples. We could limit the generation to a maximum amount of functions included in each tuple. With this approach, using all 120 vulnerable functions and limiting tuple length to be between one and three functions, we could generate $288,100$ pure vulnerable samples and naturally the same for benign samples. With a combination of both libraries and the same restrictions on size, we would generate $2,304,200$ unique samples. However, the same limitations are present in this approach as with the Cartesian product. We do not desire multiple vulnerabilities within one data sample. Another limitation of itertools is, as we can see in listing 5.11, the first functions are also used more frequently than all the others. As stated earlier, an uneven distribution over function inclusion is an undesirable distribution for us.

Using np.random.choice

Finally, let us take a look at our chosen approach for choosing functions within our generator script. The best way, to our knowledge, achieving short, unique, uniformly distributed, and single vulnerable functions for the vulnerable samples is through using python's random.choice [78]. With this library, we ensure that our script randomly selects multiple functions given parameters: number of functions to select from, how many functions we went to select, and whether or not we would like to have replacement i.e., allow duplicates. We create a list of benign functions with specified length, and a single randomly chosen vulnerable function. In addition, an index decides where the vulnerable function is supposed to be placement within the program file. This randomly generated index means that there is a degree of randomness

with regard to the placement of the vulnerable function. This randomness allows our script to create even more unique samples since the vulnerable function's placement will vary. For the benign data samples, we create a list with a specified length in a similar way. This way of using randomness and choice() we achieve:

1. Random uniform distribution of functions used in our samples, i.e., with a large enough quantity of samples, all functions are included an equal number of times.
2. Vulnerabilities have a randomized placement within a program file, which is more similar to "real world" code.
3. We generate samples with a single vulnerability.
4. We ensure that within a single data sample, no single function is called/used twice.

We argue that these four attributes allows us to generate data samples that are as real world, and as close to how we want our data to be structured as possible. Below is the mathematical formula for calculating the number of permutations from a size n set with k number of chosen elements:

$$P(n,k) = n \cdot (n-1) \cdot ... \cdot (n-k+1) = \frac{n!}{(n-k)!} \tag{5.1}$$

If we would like to generate a dataset where we each sample is three functions long and we would like to include half of each library into the set n we would be able to generate in total:

Pure benign: $P(60,3) = \frac{60!}{(60-3)!} = 205320$
Vulnerable: $P(60,2) \cdot P(60,1) = \frac{60!}{(60-2)!} \cdot \frac{60!}{(60-1)!} = 212400$

The number of possible combinations is due to sampling without replacement. With the pure benign, we sample three times. First, on the full distribution. Secondly, on the full distribution excluding the previously sampled function.

Lastly, the full distribution excluding the previously sampled functions. With the vulnerable data samples, we sample a single function from the vulnerable library as well. Therefore we only have two steps of sampling form the benign library. The formula extends as the number of functions to include increases. We have included a full code sample (A.1) in the appendix to make explicit what a generated program looks like before feature engineering.

In conclusion, not only are the data samples generated closer to "real world" code samples, but it also allows us to create a great deal of unique data samples, even with a small number of functions to write per sample. We hope that with this generation technique, we can create not only the amount of data needed but also data that has enough variance such that our model is applicable to unseen samples. In a perfect world, we would have all this data available, but we will have to make the best use of what we can create ourselves. This discrepancy between our in-sample data, which is the data we create and train on, and the out-of-sample data, which is the unseen "real world" data we want to perform classification on, is unfortunate. However, our primary focus will still be to discover the ability to do classification on data samples. If our model can perform classification on our generated data, there is a good chance that with proper training, our model might work on more real-world data as well.

5.2 Data representation & cleaning

Cleaning and creating a representation for data is usually one of the most time-consuming parts of any deep learning process. Together with data generation, these two parts of our development pipeline were both challenging and time-consuming. As aforementioned in the previous section, we elaborated a great deal on our decision-making concerning generating data and how we ended up doing it. This section is about our approach for finalizing data into a usable format for our deep learning model.

5.2.1 Representational level

There are multiple ways to represent code. Some of them are language-specific representations like Java, Python, C, Assembly, or machine code. Other possible ways are tokenization, decision trees, and even taxonomizing. The representational level was one of our initial concerns and issues to reason we discussed representational level we had some criteria we wanted to fulfill:

- First of all, we believe that code, in its logical nature, contains some structure. We believe that this structure is informative in a similar way to what we see in written and spoken languages. Therefore we want to explore the possibility of feeding code "as it is" through some deep learning model. The deep learning model we have chosen has historically been used for natural language processing. Our adoption of such a model creates a premise for us to fit and structure code in a meaningful way for a deep learning model. Since we choose to approach code "as is", we do not want to do any taxonomy or other feature extractions or manipulations in which pure code is disregarded as a feature in itself.
- Secondly, we wanted our generation and cleaning to be effective. Due to our limited hardware resources available, as discussed in the previous section, this is important. Another argument for keeping generation and cleaning at low cost is time restrictions as the more manipulation and feature extraction we do, the more time consuming our whole pipeline will become.
- Thirdly, we wanted our model to be as generalized and available as possible. If we were able to train a model that can indeed make reasonable predictions on our training and testing data, we want it to be deployable and useable for real-world data with as little overhead as possible. Therefore, we want to keep generated and finalized data as similar to a representational level available in real-world data. This similarity is important to us, as the model reusability and usefulness

are key characteristics in our validation. In other words, we want our model to be useable on real-world data with as few data manipulation prerequisites as possible. If the representational level is in such a way that new real-world examples need a lot of formatting and cleaning, we deem the model less successful.

- Fourthly, we want our data to be as compact as possible. Storing and processing large amounts of data samples is computational heavy, and even the smallest optimizations may go a long way towards efficiency.
- Lastly, out of sheer curiosity, we wanted to scrutinize a representational level currently, to our knowledge, still little explored for such purpose as ours.

With these criteria in mind, our final decision settled at using actual code as data. More specifically, we settled for assembly code, compiled with Intel syntax for the 64-bit architecture. We argue that this way of representation fulfilled our prerequisites aforementioned. Assembly is indeed a language, and we can assume that though it is not as rich as other programming languages, or spoken and written language (concerning dictionary size), it should be a suitable format for a Recurrent Neural Network. It is also relatively easy and resources efficient for us to generate data samples when we use this kind of representation level. After generating C code, we simply need to compile it down to a binary file and pull the pertinent code from it. We will elaborate further on what we consider as applicable code at a later point in this section.

We also had the option of using C source code as a representational level. However, we want to develop a model that can perform classification on executables without available source code. There exist numerous great tools for analyzing and decompiling binaries, such as the infamous IDA Pro by Hex Rays [79], Binary Ninja [80], and NSA's own Ghidra [81]. What is common for most decompilers is that they are very good at decompiling machine code and thus provide the user with generated assembly code for a binary file. However, reverse engineering machine code up to C is not yet successfully

implemented. Indeed it is a compelling argument that in the real world, the highest possible code representation obtainable from unknown binaries is assembly code. This limitation is tied to our third criterion of availability and generalization. With a model trained on assembly code, it is relatively trivial to analyze new data samples, as one would only need to extract code from the disassembly tool and feed it into the trained model.

On a side note, we also considered representing data as machine instructions as well, that is, sequences of binary numbers. We quickly discarded that idea, as each data sample would be an unnecessarily long sequence of 0's and 1's. Besides, it is more challenging to develop a deep learning model that would be able to do any learning and classification on binary data. Also, when machine code is extracted directly from the binary, a lot of compiler and architecture-specific code is included, which is not beneficial nor necessary for our purposes. If we compare the sizes of an assembly dump against a machine code dump, it is quite clear which representational level is the better choice if the intended purpose is to keep data as compact as possible. We chose to use some techniques to identify and remove what we consider inapt code in the generated data to minimize the size of each data sample.

5.2.2 Cleaning data samples:

There are some substantial challenges for our data cleaning as we, on the one hand, want to keep data as close to "real world" data samples as possible, and on the other hand, want to compress it as much as possible. If we keep every line of code from a binary, we risk including some lines of code that have no value i.e., redundant or useless information for our model and just slow down converging. Also, data samples will be much larger, and thus use more space and computational time to process for our model. Contrarily, if we remove too much code from the binary data samples, we risk losing context and create a large gap between our data samples and unseen "real world" data samples. Both scenarios directly contradict our criteria two, three, and four. We will now present below: the chosen compiler, compiler flags, which parts of the compiled binaries that have been modified or removed, and our

reasoning for our final choices.

Compiler & Flags:

We chose to use the GCC compiler [82]. Our main argument for using it is that it is cross-platform, initially made for the C programming language, has many compiler options, and GCC is the conventional compiler today for the C language family. Since it is cross-platform, it satisfies our criterion about creating a generalizable model. With the vast choices of compiler options, we can also compile our binaries as close as possible to how we envision the final structure of our data samples. We realize that binaries throughout are compiled with different flags depending on their platform, architecture, and usage. Even though this may contradict our generalization criterion, we opted to use a minimal amount of compiler flags to minimize the gap between generated binaries and "real world" binaries.

Our compiler script compiles every generated C program with the following options:

```
gcc -S -fno-asynchronous-unwind-tables -masm=intel ./fileName.c -o
    fileName
```

Listing 5.12: Compiler flags for our generator script

Each compiled binary is moved and separated into directories for benign and vulnerable data samples. Let us look at the flags, and why we chose them:

-S: This flag stops after the stage of compilation. That means that the binaries will not be assembled. The output is in the form of an assembler code file. If we were to discard this flag, we would have to add a disassembler to our pipeline (which is probably necessary for any real-world binary, but for our purposes it would only add another step to our pipeline).

-fno-asynchronous-unwind-tables:

-fno: is simply the negative form of any instructions it is linked with.

-asynchronous-unwind-tables: These directives tell the GNU Assembler to emit Dwarf Call Frame Information tags[83]. Both .cfi_* directives and .The compiler includes lxxx labels by default. The .cfi_* directives are used for unwinding information more easily and reconstruct a stack backtrace when a frame pointer is missing [84]. The .Lxxx labels indicate that the label is local to this file, so it will not conflict with the same-name labels in other files. Since we are neither interested in unwinding information nor running multiple binaries at the same time, both of these labels are redundant to us. When we combine -fno to create a negative form, we effectively ignore these directives and labels in our final compiled binaries.

-masm=intel: Generates binaries with Intel syntax instead of at&t syntax. Both syntaxes are equivalent concerning representational capability, but the Intel syntax is superior for our purposes as it has a more compact syntax. Most prominently, the use of extra characters such as '%' prepended any register used, and the '$' prepended any numerical value[85]. Since each binary contains multiple lines of assembly instructions using both registers and numerical values, we can save space and computational expenses by choosing the Intel syntax.

Even though the compiler flags removes a lot of data we deem unessential, there were still some improvements to be made. With our finalizer script **finalizer.py** we start with removing the entire prefix of each binary:

```
1 .file "test_file_3_0.c"
2 .intel_syntax_noprefix
3 .text
4 .glob main
5 .type main, @function
```

Listing 5.13: Prefix for a compiled binary sapmle

The prefix is equal in every way for each binary except the .file name, of course. Since this is present in every compiled binary, it will not help our

neural network converge in any way as it contains no variance over each data sample.

There are also other assembler directives we choose to discard to save space and keep data samples as short and concise as possible [86]. The .LC directives containing the subdirectives .string. These directives are for storing any declared string in the C program. For our vulnerabilities, the actual strings stored are not part of any buffer overflow vulnerability as they are all based on the user input to our programs. These directives serve no benefit for our classifier, and can we can remove them safely. However, we chose to keep the reference for the string directives whenever they are loaded. By keeping the references, we aim to maintain as much as possible of the code context. Listings 5.14 and 5.15 provides an example:

```
1 .LC0:
2     .string "Enter the size of input:"
3 .LC1:
4     .string "%d"
5 .LC2:
6     .string "%%%ds"
7 .LC3:
8     .string "Enter input string:"
```

Listing 5.14: Generated assembler directives

```
1 mov     rbx, rax
2 lea     rdi, .LC0[rip]
3 mov     eax, 0
4 call    printf@PLT
```

Listing 5.15: Generated assembler directive references

We chose to remove the .size directives, which are generated by compilers to include auxiliary debugging information in the symbol table. They are neither necessary for us nor necessary for the object file. .size is only meaningful when generating COFF (Common Object File Format) [87] format output. COFF is a format for executable, object code, and shared library computer files used on Unix systems, largely replaced by ELF [88] today. Another

directive we chose to remove was the .ident directive in each binary. This directive is used by some assemblers to place tags in object files. It simply accepts the directive for source-file compatibility with such assemblers but does not emit anything for it. Here is what it looks like:

```
1 .ident     "GCC: (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008"
2 .section     .note.GNU-stack,"",@progbits
3 .section     .note.gnu.property,"a"
```

Listing 5.16: .ident directive and .section directives in a binary

We also removed he .section directives. These directives are only supported for targets that support arbitrarily named sections; on a.out targets, for example, it is not accepted, even with a standard a.out section name. Again this is part of the COFF formatting for executables, and therefore we can safely discard them. The last part of the binary executables we removed was the endbr [89] instruction. This instruction stands for "End Branch 64-bit", or more precisely, Terminate Indirect Branch in 64-bit. It is an instruction used for marking valid jump target addresses of indirect calls and jumps in the program for the 64-bit architecture. Since it as well is present in all our compiled binaries, we chose to remove it.

One of the primary actions done in natural language processing is creating tokens to represent each symbol, word, or sentence, depending on the goal of the model. Thorough documentation about tokenization is found in the section about our deep learning model. However, we would like to mention our approach to creating an "optimal" vocabulary for our model. Since assembly code is a translation of C code, it is only natural that there will be many unique lines of code for each data sample. Too many lines pose a challenge. Tokenization of instructions "as they are" would create an extensive vocabulary with many single instance tokens. To mitigate this, we choose to split some of the sub instructions into atomic characters so that we create a smaller vocabulary. If we use line 5 in listing 5.19, the tokenization will create tokens:

```
"mov", "QWORD", "PTR", "-104[rbp]," and "rdi"
```

Listing 5.17: Tokenization of a line of code

Special token "-104[rbp]," would most likely be unique for the whole dataset. Therefore we choose to use several regex expressions to split these combined instructions into more generalized tokens. Continuing using this same line of code, the tokenization would instead create tokens:

```
"mov", "QWORD", "PTR", "-104", "[", "]", ",", "rbp", "rdi"
```

Listing 5.18: Tokenization of a line of code after regex

We believe that this tokenization also gives more meaning and context to the usage of these characters, and their combinations throughout all data samples.

The last tasks our finalizer script executes are labeling each dataset, splitting it into training and testing collections, and labeling each data sample to the class in which they belong. Each data sample is prepended with a label that our model will use to categorize and create label vectors for our supervised learning. Each new sample is prepended by a label, either:

- __label0__ : for benign data samples, and
- __label1__ : for vulnerable data samples,

Since we have complete control and separate directories for the benign and vulnerable programs, it is easy to use the file path as a reference for correct labeling of each sample. These labels are used as feedback by the neural network during training. During testing, these labels are used to calculate accuracy.

As we aim to feed our neural network with a single line of instructions at the time, we chose to keep the newline character '/n' to split data into desirable chunks during training and testing. We could, of course, add another token for representing a new line of instructions, but it would still result in the same number of bytes per sample. Either way, we need some way to represent a

new line of instructions, and the length of each line varies throughout the executables, as shown in listing 5.19.

```
1 push      r13
2 push      r12
3 push      rbx
4 sub       rsp, 72
5 mov       QWORD PTR -104[rbp], rdi
6 mov       rax, QWORD PTR fs:40
```

Listing 5.19: Various length of each line of instructions

During finalization, we also split all data samples into separate .txt files (Train.txt and Test.txt) for training and testing. We chose to do a split where 80% of our samples are for training. The remaining 20% are for testing. The training set will be split again into a validation set used for accuracy assessment over hyperparameter optimization. We keep the testing samples unseen for the model during training, as they will be used to measure accuracy in the final assessment. Keeping a portion of the data hidden is essential to prevent our classifier from learning classification on what is supposed to be unseen test data.

To tie the generation of data together with our chosen representation/cleaning and creation of features, we have included a pipeline diagram illustrating how data samples are created from the function libraries.

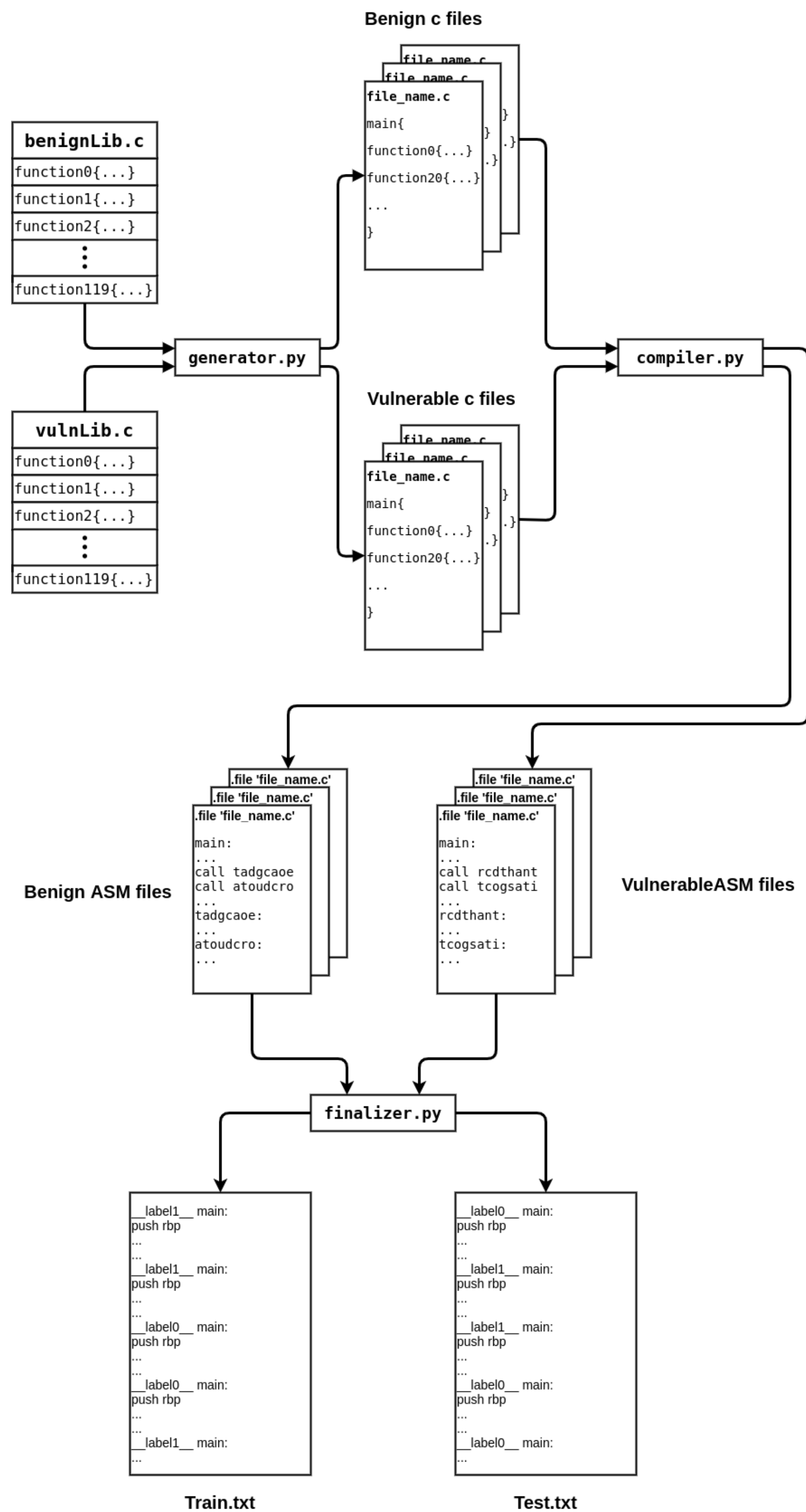

Figure 5.1: Pipeline of data creation

5.3 Architecture overview

In this section, we would like to present the technical aspects of our deep learning model. The technical aspects include an architecture overview, a summary, and an illustration of our architecture. We will also present details about our implementation and explain how we pass through our layers. Implementation details include a description of our embedding, tokenization, and prediction layer. We also elaborate on our choices concerning our problem statement and our limitations.

As previously stated, our goal in this thesis is to prove/disprove whether or not recurrent neural networks are capable of performing vulnerability detection on code files. We are proposing a simple natural language processing technique and assessing whether it is applicable and useful for classification on assembly code. As this is a proof of concept, we do not strive for or have the capacity for developing a generalized model, capable of classification over a multiflora of source code. Our goal is instead to strive for some interpretable results which can sway our judgment of said possibilities. Even though we opt for a simplified model, we still work to the best of our ability to achieve as good results as possible.

Compared to the work revised in the literature review, our model is quite simple. We do not compare our model against other researchers' work as we opt for a minimalist approach. We believe it is entirely possible to surpass our results with deeper networks and more sophisticated hardware. We ended up creating a fairly shallow and "simple" architecture for our experiments, which gives us some advantages and disadvantages. Training a shallow model is far less time consuming than a deep neural network but usually at the cost of lower performance. For our hardware capacities, a deeper network would be well-nigh impossible concerning time and computational constraints. Thus, when designing our architecture, we aimed for these features:

1. A computationally lightweight architecture.
2. Ability to differentiate between vulnerable and benign binaries.

3. Ability to consider both local and global context while examining the entire binary.

To achieve 1), we created a neural network with an embedding layer, two hidden layers of LSTMs followed by a fully connected layer, and a single output neuron for the network's prediction. Compared to other successful models, this is very lightweight, and training/testing such an architecture is computationally relatively straightforward. Success concerning feature 2) is revealed in the results section. Feature 3) is achieved with our temporal dropout and transformation of tensors. Presented below is a simplified figure of our architecture with detailed explanations about each part of it.

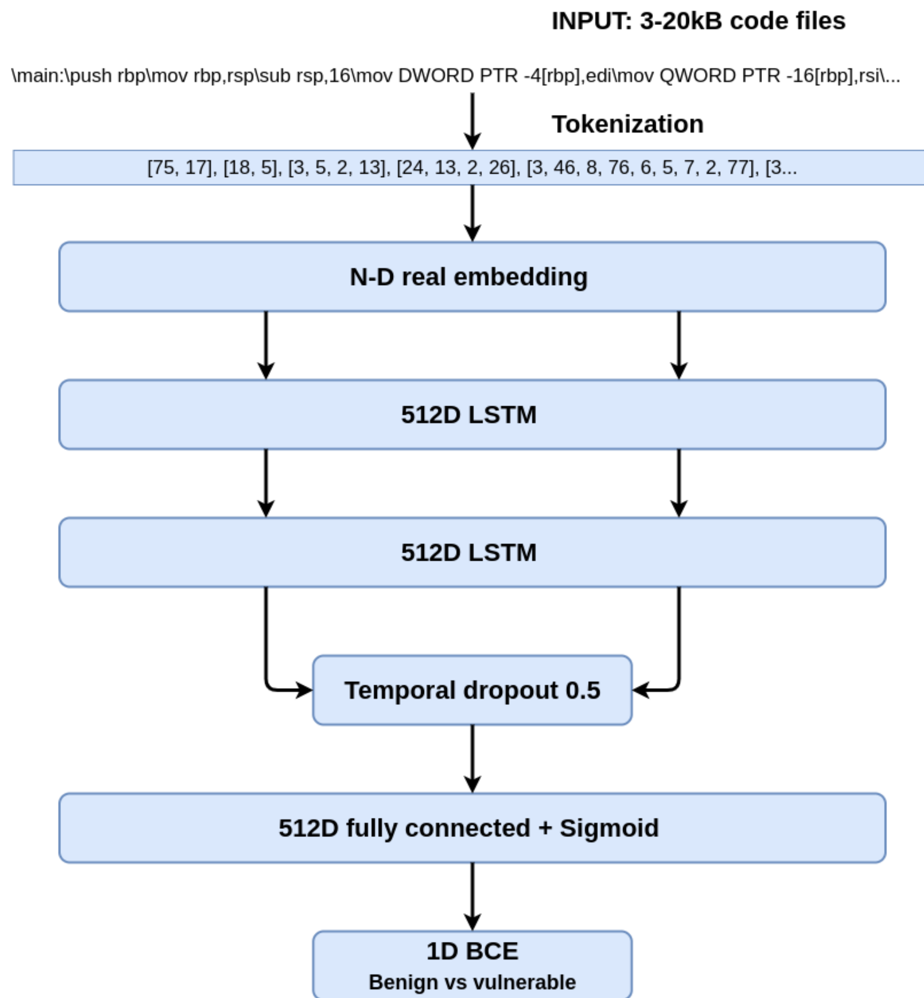

Figure 5.2: Architecture stack

As explained in earlier sections, input data are whole assembly text files stripped of what we have defined as redundant or non-critical information. All these files are read into our program code, which further translates the assembly language to something interpretable for our machine learning model. The standard way to transform textual data into learnable vectors in deep learning is a technique called **tokenization**.

Tokenization:

Tokenization is a standard Natural Language Processing (NLP) task where

each sentence is split into individual tokens, which in our case consists of words, numbers, and single sign characters used in the Intel assembly syntax. For this process, we use NLTK (Natural Language Toolkit) [90], as it has one of the faster tokenizers. All words in the data collection are stored into a dictionary where words are mapped to their number of appearances, creating a vocabulary of the whole training file.

```
# Dictionary mapping of all tokens in our training file
words = Counter()

for i, line in enumerate(train_file):
    for word in nltk.word_tokenize(line):
        words.update([word])
```

Listing 5.20: Dictionary mapping

When we later transform training and test data into "sentences", we want to account for any unseen words and padding. Therefore, we add them to the vocabulary as well. These will occupy the first two indices in our dictionary. We then create an additional dictionary with mapping **word2idx**. This mapping is used for doing a **word2vec** transformation of our data.

```
# Sorting the words according to the number of appearances
# with the most common word being first
words = sorted(words, key=words.get, reverse=True)
words = ['_PAD','_UNK'] + words
word2idx = {o:i for i,o in enumerate(words)}
```

Listing 5.21: Word to index mapping

Every word in each training and testing sample is converted into integers corresponding to their index in our vocabulary.

```
# Looking up the mapping dictionary and
# assigning the index to the respective words
for i in range(len(training_data)):
    for j, sentence in enumerate(training_data[i]):
        training_data[i][j] = [word2idx[word] if word in
        word2idx else 1 for word in sentence]

```

```
8 # For test data, we have to tokenize the sentences as well
9 for i in range(len(test_data)):
10     for j, sentence in enumerate(test_data[i]):
11         test_data[i][j] = [word2idx[word] if word in
12         word2idx else 0 for word in sentence]
```

Listing 5.22: Assigning index to each word

After tokenization on each sample is completed, we zero pad each sample to match the longest sample over the whole collection. Our data samples are dimensionally more complex than samples from "regular" natural language processing. This increased complexity is due to the fact that in a natural language, a basic unit is usually a word or a letter, while in our assembly code, we are operating on basic units as lines of code. Operating on sentences of code results in an additional dimension for our data samples, which must be accounted for and transformed in a meaningful way.

Embedding layer: Including an embedding layer to our neural network is crucial with regard to computational efficiency. This layer allows us to turn positive integers (in our case indexes) into dense vectors of fixed size. This is the **word2vec** operation we mentioned earlier. Embedding in deep learning is the act of representing or encode a sentence. Input dimensions to our neural network during a forward phase will look like this:

```
[batch_size, codefile_size, line_size]
```

Listing 5.23: Input dimensions

Looking at a single line of a single sample within a batch, we have a raw line of code:

```
[mov DWORD PTR -4 [ rbp ] , edi]
```

Listing 5.24: Raw line of code

Which through our tokenization is represented as a numerical vector:

```
[3, 46, 8, 76, 6, 5, 7, 2, 77]
```

Listing 5.25: Raw code represented as numerical vector

The embedding now creates an embedding matrix based on this numerical vector, and how many "latent factors" we assign to each index. In our implementation, we chose latent factors to be equal to the longest line of code in our data collection. The embedding matrix for this particular vector will look like this:

$$\begin{bmatrix} [0.380, & 2.003, & -0.643, & 1.288, & 0.295, & 0.160, & -0.023, & 0.447, & -0.371] \\ [0.418, & -1.453, & -1.167, & 0.283, & -0.738, & 0.810, & 0.061, & 0.959, & 1.061] \\ [0.033, & -0.692, & -1.704, & 0.359, & 0.668, & 0.683, & 0.101, & -0.531, & -0.776] \\ [0.508, & 0.489, & -0.651, & 0.382, & 0.172, & -1.105, & 0.485, & 0.707, & 0.126] \\ [0.423, & 0.488, & -1.053, & -0.474, & 0.471, & -1.450, & -1.236, & -0.107, & -0.739] \\ [0.363, & 1.256, & 0.557, & 1.702, & -1.272, & -0.409, & -0.857, & 1.543, & 1.404] \\ [-0.115, & -0.451, & 0.406, & 1.498, & 0.873, & 0.285, & -0.193, & 0.572, & -0.999] \\ [0.656, & -1.017, & 0.087, & -0.717, & 0.901, & -0.457, & 2.186, & 1.035, & 1.712] \\ [-0.711, & 0.442, & 0.589, & -0.682, & -0.491, & 0.804, & -0.217, & -0.428, & 0.090] \end{bmatrix}$$

Since each element/number in the tokenization is represented as a numerical vector, we had a decision concerning dimensional representation. LSTM layers will not accept the current (4-dimensional representation): [batch_size, codefile_size, line_size, embedding_dim]. To transform our representation, we had two options: do some form of mathematical operation on each vector to reduce it into a single decimal i.e., some reduction, or flatten the matrix into a list. Each option is viable and represents data in a slightly different way. With a reduction, representation would be more focused on a line of code as a whole. The challenge here is to find some function that reduces our vectors into something tangible and useable for our neural network. Instead, we opt to flatten the matrix into a list. With this approach, each instruction and its context will be the focal point for the neural network. We believe that this is just as useful as the foremost approach. The negative aspect

of this approach is that training-time will certainly increase compared to a reduction operation. By flattening the matrix, we do not reduce the number of values for a given line of code. After flatting the embedding matrix, a line of code is represented in a meaningful way for our neural network.

```
def forward(self, x, hidden):
        batch_size = x.size(0)
        x = x.long()
        embeds = self.embedding(x)
        embeds = torch.reshape(embeds, (batch_size,
        longest_sample, embedding_dim*embedding_dim))
```

Listing 5.26: Embedding in the forward phase

Each embedded line of code is fed to our first layer of LSTM cells, which means that a single line of code at the time is fed into the network during the forward phase.

```
lstm_out, hidden = self.lstm(embeds, hidden)
```

Listing 5.27: Embedding fed into the LSTM layers

The output from our first LSTM layer is forwarded to our second LSTM layer. After the second layer has finished its computations, a dropout regularization, with a probability $p = 0.5$, is performed.

```
out = self.dropout(lstm_out)
```

Listing 5.28: Temporal dropout

With our temporal dropout after our LSTM layers, one way to interpret this network's function is that the LSTM layers are capable of recognizing local indicators of vulnerabilities, and the dropout followed by a fully connected layer assesses the relative strength of those indicators throughout the file and recognizes significant global combinations.

After the temporal dropout, we connect outputs to our final hidden layer, which is a 512 dimensional fully connected layer using the sigmoid activation function, described in the background section.

```python
out = self.fc(out)
```

Listing 5.29: Output from the LSTM layers is fed into the FC layer

From these outputs, a loss is calculated by using the binary cross-entropy loss before the backward phase begins.

```python
loss = criterion(output.squeeze(), labels.float())
loss.backward()
```

Listing 5.30: Loss calculated by criterion (BCE)

We believe that this neural network architecture fulfills our desired features, and serves as a basic model for assessing our problem statement. As our computational power is relatively limited, this lightweight network accommodates our hardware architecture in a way that allows us to conduct additional experiments.

5.4 Summary

In this chapter, we have elaborated on our approach for generating data, how we have performed feature engineering on our binaries, and finally presented an overview of our architecture. For each of the subsections, we have displayed a thorough discussion and arguments for choices made with regard to data, features, and final architecture concerning our research goal. We have presented which vulnerable system-calls we have focused on and given code samples of generated functions. In the data generation section, we have provided a step-by-step walk-through on how we generate data samples from our function library. We also gave an in-depth description of how features are extracted and written to file. The aforementioned sections were tied together with a pipeline diagram exhibiting the whole process. We concluded this chapter with a presentation of our chosen architecture and the inner workings of our neural network code.

Chapter 6

Results and Evaluation

In this chapter, we present our experiments, results, and evaluations of them. We have been carried experiments out over an array of different parameters and a variety of generated datasets. We provide detailed descriptions of said parameters, datasets, and achieved results for these. This chapter is divided into subsections, encompassing the exploration of specific criteria we would like our neural network to achieve. By scrutinizing one desired attribute at a time, we enable a better basis for us to perform correct assumptions. We have, in total, conducted five experiments with a range of hyperparameter configurations. Each model has been run several times for validation purposes. We base our evaluation and analysis entirely on these experiments, their parameters, and datasets. Experiments are listed in order of complexity, where we either decrease the variance over data samples or increase the number of possible permutations of samples. Our goal is to discover and gain insight into the capabilities and limitations of our neural network.

To begin with, we would like to state that this work was extremely challenging concerning data availability and hardware limitations. We were somewhat anxious from the starting stage of this master dissertation but excited about what kind of results we could expect to see. From the get-go, we held a positive skepticism towards our research.

Optimizing a neural network is no easy task. There is a multifold of hyper-parameters to tune and tweak. There are entire research fields dedicated to optimizing and unveiling optimal configurations of every single one of them. To give a perspective of the possibilities, let us look at one of the most common parameters: learning rate. At extremes, a learning rate that is too large will result in weight updates that will be too large, and the performance of the model (such as its loss on the training dataset) will oscillate over training epochs. Oscillating performance is said to be caused by diverging weighs. A learning rate that is too small may never converge or may get stuck on a suboptimal solution. The consensus is that a reasonable learning rate for gradient descent learning should fall between 0.1 and 1e−5 [91], which is a relatively broad range when we consider the possible decimal number in between extremities. There are also techniques such as scheduled and dynamic learning rates, which change the learning rate according to predicted necessity and by actual measurement of cost during runtime. An example is the ADAM optimization algorithm described in the background section.

We generated different collections of training and testing data to assist us in proving/disproving and scrutinize our problem statement. We varied the datasets by their size, function distribution, and experiment with a perfectly imbalanced dataset (single class dataset). The latter is useful in dissecting whether weights are updating accordingly to loss and that the model does not overshoot and oscillates too much around a zero loss. We chose to gradually increase the size and complexity of our data collection to unveil our small scale RNN LSTM architecture's capacity and accuracy. The central premise for our data collection is that we must keep the number of generated samples below the total amount of permutations available.

To make this clearer, let us imagine that we would like to create a collection of data, where each data sample consists of two functions. Let us also limit the collection to only draw from the first ten benign functions. In total we would then be able to generate 90 unique samples with our generator script:

$$P(10, 2) = \frac{10!}{(10-2)!} = 90 \tag{6.1}$$

We are guaranteed to introduce some duplicates to our collection if we were to generate more than 90 data samples. This possible duplication of data samples is a problem. The smaller the difference between the number of generated samples and possible permutations, the higher probability of a duplicate sample in the data distribution. Duplicates, especially ones shared in both training and test data, results in memorization rather than a generalization over features.

By keeping the total amount of data samples well below possible permutations, we guaranty unique samples in our training and test data.

In the following results section, we will present "subgoals" that we would like to prove/disprove with configurations, documented experiments, and analytics of both data and results. We have developed these experiments in a natural progression from the most basic experiments to the more advanced ones. Before we present our results, we would like to explain how we measure and log results. First of all, we are logging results after a predefined number of "steps", i.e., number of gradient updates. Our models run for a certain number of predetermined "epochs", i.e., full cycles through the training data. Batch size is a term used in machine learning and refers to the number of training examples utilized in one iteration. We plot our results into a figure for visualization where we table CCR (Correct classification rate [0,1.0]) and BCEL (Binary cross-entropy loss). We plot the progress of both the training and development set (validation set). For some of the experiments, we have chosen to document some failed configurations as well. Our rationale is to convey how small changes to configurations can have a drastic impact on

performance and results. Keep in mind that for the "simpler" experiments, our network was able to converge for most configurations. In these cases, parameters mostly impacted the amount of time the neural network needed to converge to a "sufficient accuracy".

For some of our experiments, we predefined a sufficient accuracy, or rather loss on the development set before the model is evaluated on the test data. In some cases, we require the loss to be 0.001 or less. This loss threshold is arguably too strict and could make the necessary training time for our neural network too high. However, we have adjusted this threshold for more complex data. In all experiments, we let the neural network run for an "unlimited" number of "unsuccessful" epochs and used our threshold on the validation set to decide when to stop training and start measuring actual accuracy on the test set. In practice, this means that we allow a certain number of epochs without improving accuracy over the development set before we terminate the experiment. In cases where there is no improvement in accuracy, even after several hundred epochs, we conclude that our hyperparameters were not optimized, and we terminate training.

For each experiment, we list multiple options for hyperparameters. We have run models with each possible combination of parameters several times to validate results and configuration. By and large, the successful hyperparameter options are the ones documented thoroughly and used for evaluation. Choosing the right hyperparameters is quintessential in almost all deep learning tasks.

We would like to draw attention to, and state, that we are only concerned with **Learning Hyperparameters**. Specifically, initial learning rate and batch size. These two parameters have a significant impact on performance and the rate of convergence for a neural network.

6.1 The network can classify a single class

First, we would like to verify if there are any possibilities of our network to learn anything from this representational level. We conducted a simple experiment with an entirely imbalanced dataset i.e., the entire collection of data consists of one class of data samples. It was essential for us to ensure that weights were updating correctly according to the learning rate and that the network would be able to learn some features in the data.

```
Parameters: generator.py 100 2 12 B
Function range: Benign: B[0...11]
Permutations: 132
Batch size: [2]
Learning rate: [0.0005, 0.001]
```

Listing 6.1: Configurations.

The dataset is a small set of 100 benign data samples, which is well below the 132 permutations. We chose the batch size to fit the number of samples in the validation and testing set. As this experiment is trivial, we did not bother to test a range of batch sizes or learning rates.

Figure 6.1: Single class classification, LR:1e−3.

We trained two models on this dataset, one with a learning rate of 5e−4 and one with 1e−3. Both achieved the same accuracy. Figure 6.1 is a plot of our model's progress with the most aggressive learning rate. As we can observe, the model was updating its weights and quickly learned a 100% accuracy for a pure benign dataset. The model converged to a loss of 0.0000 and a 1.0 CCR on the development. The convergence happened after about four steps into the first epoch. The fast convergence is not surprising, yet promising as it validates our architecture and that there are learnable features directly from assembly language. To be fair, this test is more a test of the network itself rather than the networks' ability to learn from our chosen features and representation level.

6.2 The network can detect a single vulnerability

To further test our network, we introduce negative or vulnerable data samples as well. We decided to generate both benign and vulnerable data samples from the same subset of functions. This distribution results in quite a low variance in the data collection concerning the benign part of both classes. For the vulnerable samples, we programmed the generator to draw a single predefined vulnerably function from the vulnerability library. We position the vulnerability at the very start of the generated programs. Even though this is a simplified approach and is not representative of real software, it is a natural step towards more complex data. The number of generated samples are well below the total amount of permutations possible, so there are no exact duplicates across training or testing data.

```
Parameters: generator.py 2000 3 15 B & generator.py 2000 3 15 V
Function range: Benign: B[0...14] Vulnerable: B[0...14] + V[119]
Permutations: 2730 + 2730 = 5460
Batch size: [20,40,80, 100]
Learning rate: [0.00025, 0.0005, 0.001, 0.002]
```

Listing 6.2: Configurations.

Figure 6.2: Single vulnerability classification, LR:5e−4, BS: 80

Figure 6.3: Single vulnerability classification, LR:1e−3, BS: 80

For this experiment, we trained models with a total of 16 different configurations. This experiment is arguably the first real "test" of our network's potential to do vulnerability detection. Even though the dataset is quite limited by only using 15 functions from the benign library and a single vulnerability, we can observe that we need a few steps before we start seeing progress with accuracy and classification rate. The two most successful configurations are observed in 6.2 and 6.3, where the latter outperformed the former. Both models were able to correctly classify all data samples in the testing set, reaching a 1.0 CCR on the validation set after 34 and 32 epochs, respectively. From the two figures, we can also observe that the learning rate equal to 1e−3 has a faster convergence (1280 steps versus 1360 steps) and is also far more stable. We can draw this conclusion based on the fluctuations in figure 6.2.

We have not included the less successful models with consideration to space. We will, however, state that for smaller batch sizes, such as 20 or 40, our model used an increased amount of computational time before achieving successful classification. Likewise, with a very modest learning rate, the model struggled to converge. On the other end of the scale, the highest learning rate prevented our model from reaching any reliable accuracy before termination. We will not speculate whether on not this configuration eventually would reach a sufficient CCR. These results are quite promising as they prove that our neural network, given the right parameters, is indeed able to differentiate between two classes.

6.3 The network can detect multiple vulnerabilities

In this context, when we are talking about multiple vulnerabilities, we refer to data samples generated from a set of vulnerable functions, not just one in the previous experiment. In this experiment, we have included a range of vulnerable functions combined with a subset of benign functions. This way, we can further evaluate our network's ability to classify over increasingly complex datasets. From a computational standpoint, classification over samples where the vulnerable function varies is far more complicated than recognizing a single vulnerability. To make an analogy, in our previous experiment, we can imagine a classification task between two arbitrary objects. When we restrict our negative samples to a single vulnerability, we are assigning a "single" set of features that are unique to one class. In this multiple vulnerability experiment, we try to negate this by allowing each negative sample to draw from a set of vulnerable functions.

```
Parameters: generator.py 2000 3 15 B & generator.py 2000 3 15 V
Function range: Benign: B[0...14] Vulnerable: B[0..14] +
   V[105...119]
Permutations: 2730 + 3150 = 5880
Batch size: [20,40,80, 100]
```

```
Learning rate: [0.00025, 0.0005, 0.001, 0.002]
```

Listing 6.3: Configurations.

Figure 6.4: Multiple vulnerability classification, LR:5e−4, BS: 80

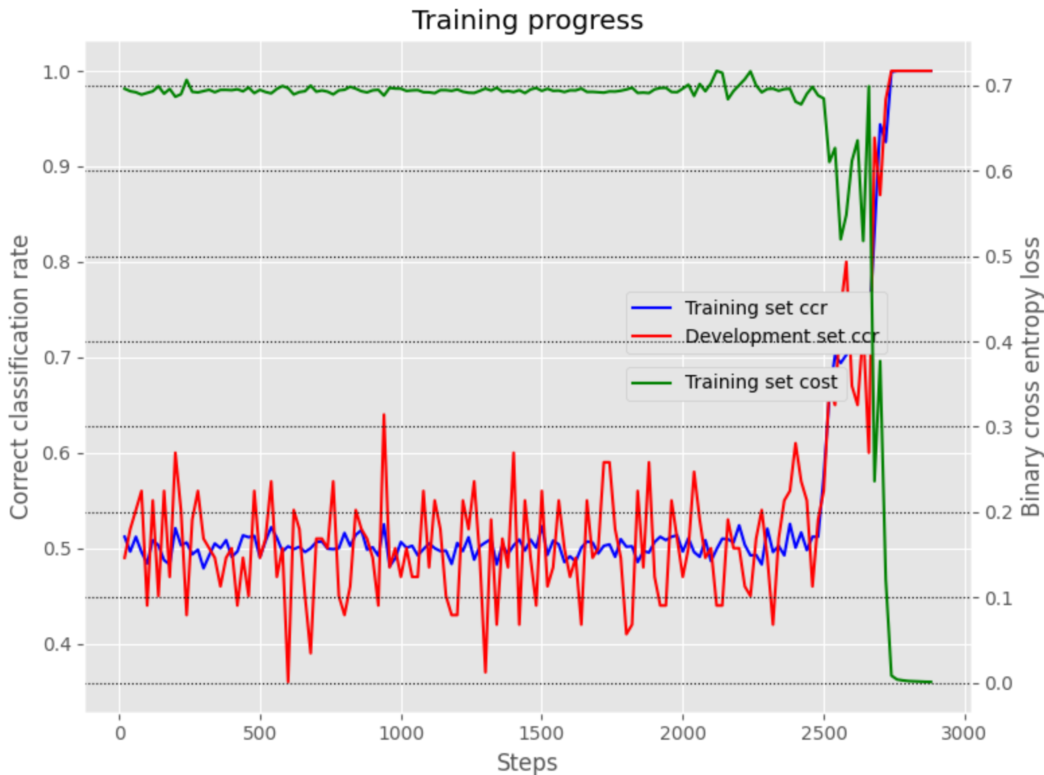

Figure 6.5: Multiple vulnerability classification, LR:1e−3, BS: 80

We continued to experiment with a range of parameter combinations, and it seems like for the dataset of 4000 samples in total, a batch size of 80 is optimal. We can observe from both figures 6.4 and 6.5 that we were able to achieve a 1.0 CCR and a very low loss of 0.001 over the validation **and** test set. What we can deduce from this experiment is that our neural network inhibits the capacity to classify without error over a subset of functions. We would like to point out the differences in the computational effort required to reach such a state. Observe how a learning rate of 5e−4 requires the network to train for over twice the number of steps compared to a more aggressive learning rate.

Similar to our previous experiment, some configurations failed to achieve desired results before termination. We allowed our models to run for an increased amount of "failed" epochs before termination, yet most models

failed. Our configuration of learning rate 1e−3 and batch size 40 was able to reach the desired CCR. Convergence was very slow, and the model did not fulfill our requirements before 20,000 steps into training. We believe that this experiment generalizes over other similar experiments where ranges of functions are applied.

6.4 The network can differentiate between vulnerable and benign counterparts

To aid us in determining our network's sensitivity concerning low variance data, we set up this experiment where we only include functions using the same vulnerable system call. We achieve this by only including vulnerabilities from a single type of system-call, in this experiment, **fgets**. For the benign data, we only allow the repaired counterparts of these vulnerable functions. These functions are incredibly similar in code, where only one line might differentiate a benign from a vulnerable sample. All of the benign functions included here also use **fgets**; however, in a safe and controlled way. We chose to keep this experiment relatively small with samples consisting of three functions and generate 200 samples of both classes. As previously we conducted, a series of experiments with different configurations.

```
Parameters: generator.py 200 3 8 B & generator.py 200 3 8 V
Function range: Benign: B[40...48] Vulnerable: B[40...48] + V[90,
   91, 92, 94, 96, 97, 98, 99]
Permutations: 336 + 336 = 672
Batch size: [2, 5, 10]
Learning rate: [0.000125, 0.00025, 0.0005, 0.001, 0.002]
```

Listing 6.4: Configurations.

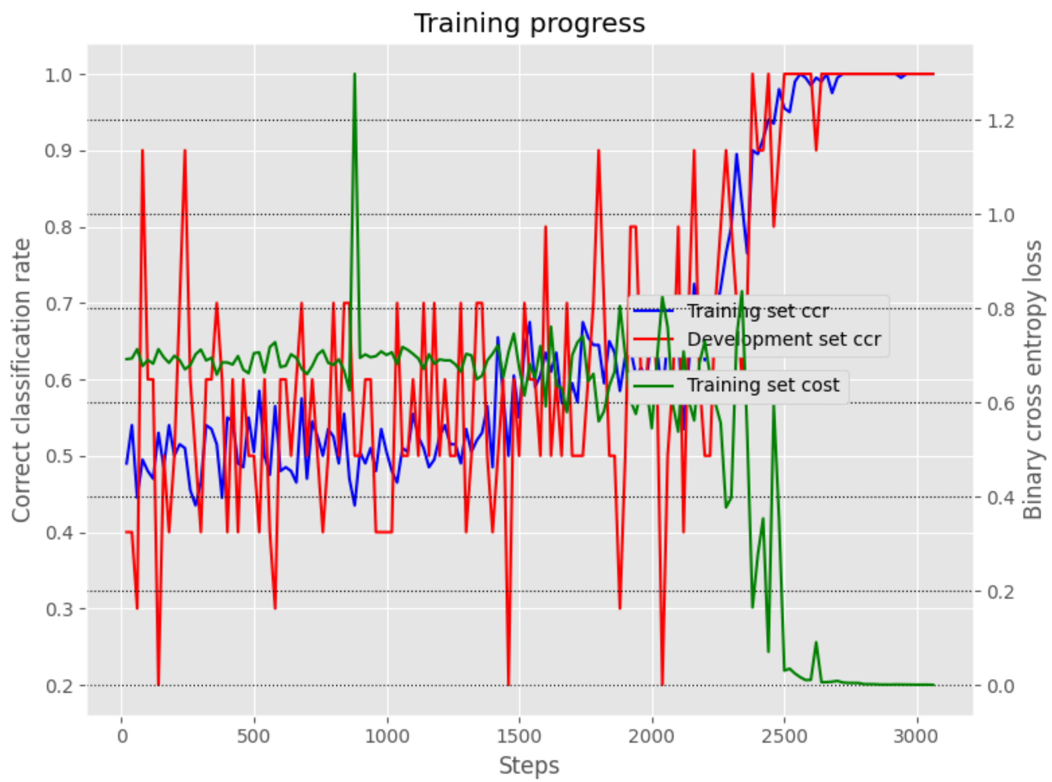

Figure 6.6: Vulnerable and benign counterparts, LR:1.25e−4, BS: 10

Figure 6.7: Vulnerable and benign counterparts, LR:2.5e−4, BS: 10

From figure 6.7 and 6.6 we can observe that our experiment was successful. These two learning rates were the only successful. Anything above 2.5e−4 often got stuck at and resulted in a 0.5 CCR at termination. We believe this has to do with the extremely low variance in the data samples. The data samples are simply too similar for an aggressive learning rate such that the approximation will alternately over- and undershoot. In some cases, the model would achieve sufficient accuracy with a 5e−3 learning rate, but on average, it is far too unstable. In general, a batch size of 10 seems optimal for this particular experiment as smaller batch sizes simply increased training time and achieved the same CCR and loss.

Comparing the two plots above, we can see that a lower learning rate, in this case, is beneficial. The model, more or less, reduces training time by about 40%. We also did some experiments where we limited the number of

total epochs to 200. This configuration allowed us to measure the loss values on the test set after an equal amount of training. It seems like the lowest learning rate is indeed the most successful and achieves the lowest loss on the test set.

The only successful configuration was using a learning rate of 1.25e−4 with a batch size of 10, which yielded a 95 % accuracy and a relatively low loss of 0.176 on the test set. With any other batch size than 10, the model could not improve accuracy on the development set, even after allowing 200 consecutive epochs without any improvement. Even though both of the lowest learning rates achieved 1.0 CCR, the loss is far lower with a 1.25e−4 learning rate. These findings signify that a lower learning rate is preferable for low-variance data. We are quite happy with the results for this experiment as it proves our model is sensitive enough to differentiate between highly similar data.

6.5 Quantitative experiment

In our last experiment, we would like to test our model performance and accuracy on a larger dataset drawn from an even more extensive range of functions. As this experiment is quite substantial, we limited ourselves concerning configurations. We set up our models with previously successful hyperparameters. E.g., a relatively conservative learning rate and a batch size enabling a good portion of data samples to be feed together through the network before backpropagation.

The idea behind this experiment is to test the network's ability to learn features from a small subset of possible binaries. The benign functions included are various functions, which are **not** part of the repaired vulnerabilities. That means that the benign functions are quite different from the vulnerable functions. The reason behind this choice is that software programs generally contain all kinds of functions and supporting functionality. We strive to replicate this by concentrating the buffer sensitive operations to a single function, which we argue resembles "real world" software. Just to be clear, the negative samples also draw from the same benign function distribution

as well as the full vulnerable function library.

```
Parameters: generator.py 5000 3 63 B & generator.py 5000 3 120 V
Function range: Benign: B[57...120] Vulnerable: B[57...120] +
    V[0...120]
Permutations: 238266 + 468720 = 706986
Batch size: [100, 200, 250]
Learning rate: [0.00005, 0.0001, 0.0002, 0.0005]
```

Listing 6.5: Configurations.

As we can observe from this configuration, we have only covered about 1.5% of the possible permutations of data samples. As before, we chose to use a single vulnerability randomly placed within the binary in an effort to make our model learn from context, rather than position. We let the models run for at most 800 epochs or until sufficient loss on the validation set.

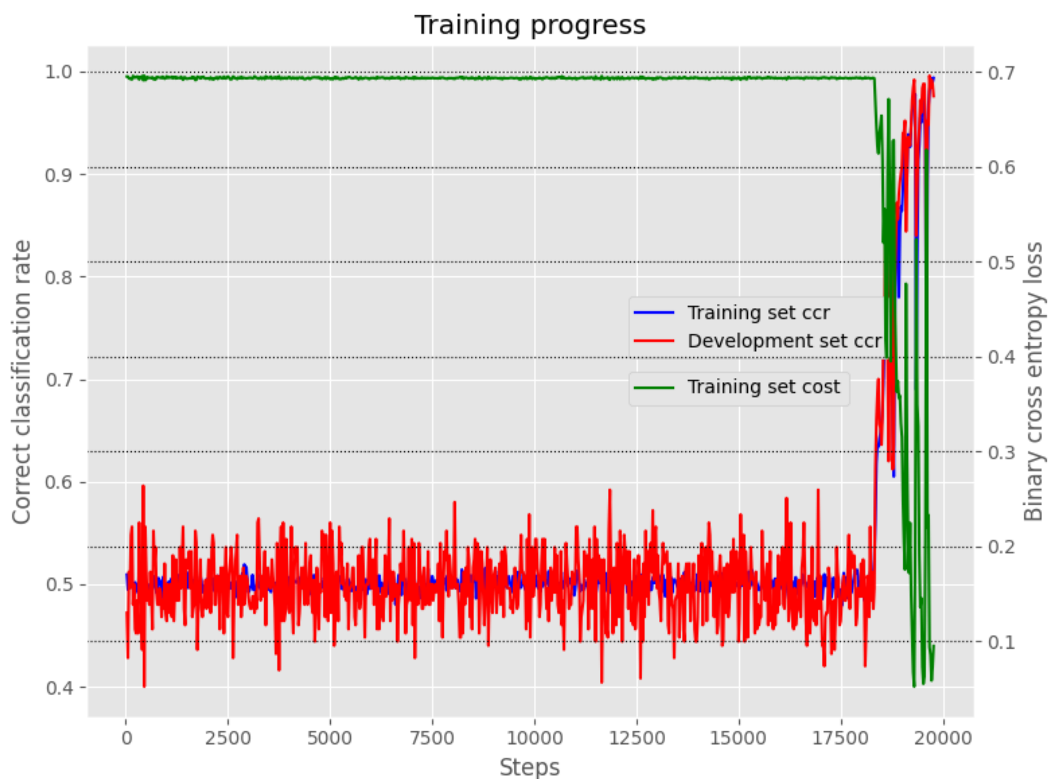

Figure 6.8: Quantitative experiment, LR:5e−5, BS: 250

From figure 6.8, we can observe a quite distinct horizontal line for the training set cost. This stagnation persists up until about 18,000 steps, which corresponds to about 600 epochs. For this particular model, we utilized the lowest learning rate of any experiment. At the point of evaluation on the test set, the model achieved an impressive 99.30% accuracy and 0.040 loss.

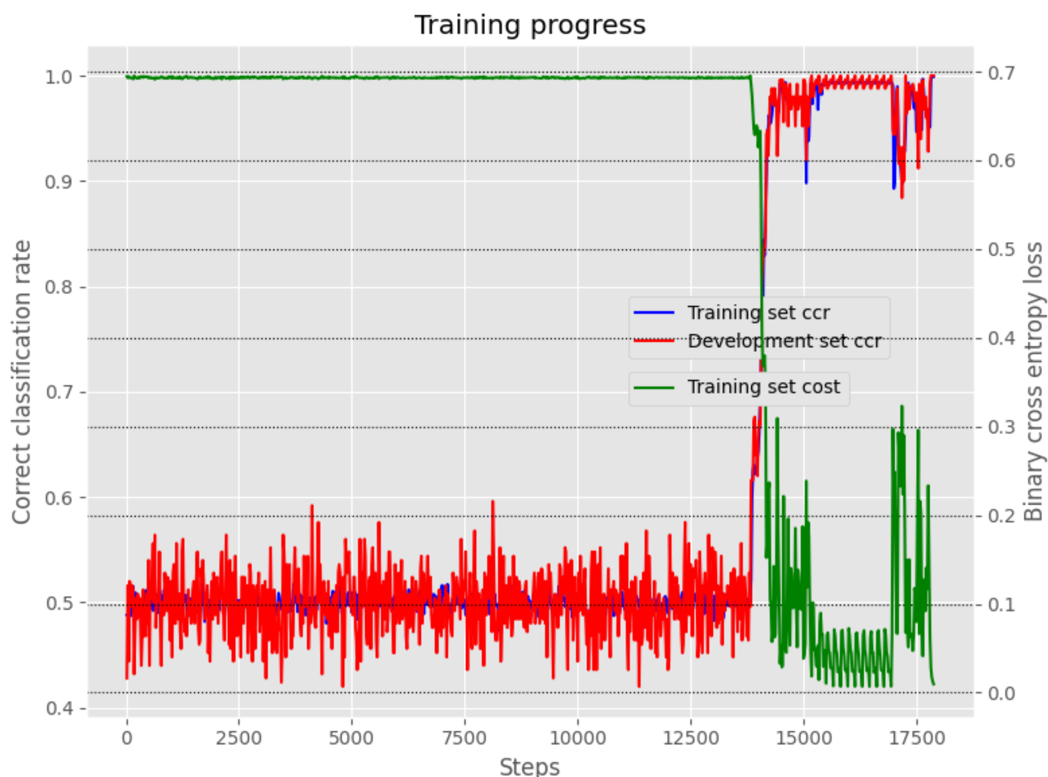

Figure 6.9: Quantitative experiment, LR:1e−4, BS: 250

Even though the model of 6.9 performed best of all configurations in this experiment, a similar stagnation is observable. The model shows a relatively stagnated learning phase for 500 epochs before improving its classification score on the validation set. This model, with a higher learning rate, achieved better and more stable results. At the time of evaluation, the model achieved an accuracy of 99.90% and a loss of 0.0013 on the test set.

We believe that the slow and almost non-existent improvement over numerous epochs is due to the fact that the classes are practically homogeneous. We can justify this statement by observing that even learning rates as low as 2e−4 were too aggressive. Only the two lowest learning rates allowed the network to converge to an acceptable loss within the time frame. Interestingly, at some point, right before reaching a sufficient loss, the model started to deteriorate. We believe this is due to the learning rate, albeit relatively small, is too

granular in the final steps. The "aggressive" learning rate makes the neural network approximate in the direction opposite to the gradient. However, since we are using the Adam optimizer, the learning rate is adjusted, and the model can correct itself over time.

We realize this experiment was very sensitive to hyperparameter tuning. Only a batch size of 250 samples yielded a model that achieved a satisfactory result within the limited training time. It is interesting to observe that we had to apply the lowest learning rates throughout our experiment to achieve any results for this particular assessment. We are, however, very pleased with the experiment, as it suggests that our neural network architecture is indeed capable of inferring meaningful patterns form more extensive amounts of data. We can also derive, at least with some certainty, that our neural network can indeed learn from a small subset of seen samples drawn from a much broader distribution.

6.6 Summary

We want to conclude this chapter with an overall evaluation of our experiments. We have already evaluated the experiments individually; now, we would like to interpret what we can prove and observe from the experiments wholly. First of all, we are very enthusiastic about our network's ability to do successful classifications in our experiments. We have seen success in some form in all our experiments and explored different learning rates, batch sizes, and number epochs we allow our neural network to train. Several observations are made throughout our experiments as a whole.

First and foremost, we can observe that the neural network is sufficiently capable of differentiating between classes of data, even when the variance between data samples is minimal. This observation especially holds true for **experiment 6.4**, where the difference between classes can be as subtle as a single if statement. Successful implementation of a vulnerability detector based on machine learning techniques needs to be sensitive to details and be able to differentiate on subtle flaws in code. This finding is very encouraging towards our hypothesis.

A second observation is that the neural network is relatively fast at converging to sufficient accuracy and loss, given the correct configurations. Generally speaking, training of deep learning models can take days, weeks, or even months on highly sophisticated hardware. We were inclined to keep the architecture shallow and the number of training samples within a reasonable limit, which has allowed us to run our models over different configurations. It is good to know that features are learnable within a reasonable time-frame even though this is not a deep neural network.

We can also observe that our neural network is fully capable of handling a decent number of data samples. In **experiment 6.5**, the neural network was processing $10,000$ data samples and able to converge within an hour of training. From that very same experiment, we can also observe that the neural network was able to do correct classification on a dataset where the distribution of generated samples was substantially smaller than the number

of permutations.

From a critical point of view, our experiments revealed that for the more complex assessments, the neural network is increasingly sensitive to hyper-parameter configurations. In other words, an optimal choice for batch size and learning rate is increasingly essential in order to achieve good results. E.g., as stated in the individual evaluations, a too small batch size will increase training time, and a too aggressive learning rate will make the network oscillate. This observation holds for both efficiency and effectiveness of training. There may be other learning rates and batch sizes that could work just as well or even better than the ones we have explored. This fact is, of course, extendable to the architectural choices we have made for our neural network. The architecture choices refer to the **Model Hyperparameters**, which we are not concerned with tuning during our experiments. We have clearly stated our limitations and possible extensions in the upcoming future research chapter.

We are aware that it is unfortunate that we are not able to experiment and tune more parameters for our models and let them run for longer. However, we believe that we have performed a sufficient exploration of each experiment. We will admit to a lack of experimentation on large data samples. We had to account for our limitations and scope for all our experiments when deciding the size limit for each data sample. With access to additional resources, we would have run our model on supplementary hyperparameter configurations and possibly created additional experiments.

Our experiments have succeeded in underpinning and providing us with a base upon which we can make conclusions. We have also gained additional ideas about possible extensions of our work.

Chapter 7

Further research

This chapter will present developments that, with more resources, could be explored in the future. We will present them in an increasingly complex order where we start with the most straightforward implementations and conclude with the most challenging ones. As our field of research is relatively uncharted, we suggest a plethora of possible extensions. First of all, our architecture could potentially be more complex by adding LSTM- or other layers. We also see the potential for adding an attention mechanism that would allow us to direct focus and greater attention to certain factors (e.g., vulnerable system calls) when processing input data. We also acknowledge the limitations of our dataset and would like to refer to the data generation section for a thorough discussion on these.

Deeper network

We acknowledge that our computational capacities are quite limited. Thus we are not able to conduct as extensive experiments as we would have liked. An intuitive way of extending our work would be to add consecutive layers of LSTM cells (or other layers) to achieve a deeper neural network. Consequently, the neural network would be better suited to learn features at the cost of training time. It would also be interesting to increase the individual size and number of data samples in a dataset and examine results and performance on both a shallow and deep recurrent neural network.

Add different types of layers

Another interesting extension of our work would be to add different types of layers. Since we are only using two LSTM layers and a single FCC, it would be exciting to experiment with supplementary layers throughout the architecture stack. As elaborated on in our chapter about feature engineering, each sample is quite extensive already, even for simple code files. An interesting approach would be to use convolution [92] and pooling layers to compress input vectors and feed their output into a stack of LSTM layers. Naturally, it would also be interesting to experiment with a different architecture stack altogether.

Attention mechanism

There is much exciting research about using attention mechanisms [93] in sequence transduction. Complex recurrent or convolutional neural networks by in large dominate this area of NLP. Research states that simple networks with attention mechanisms may outperform the standard approaches [94]. In short, the idea behind an attention mechanism is that it allows the network to map essential and relevant parts from the input sequence, in our case the instructions, to the output sequence, and assign higher weights for these essential sequences. By applying this mapping, the neural network dra-

matically increases its output prediction accuracy. This accuracy might be immensely helpful towards successful vulnerability detection as a single line of code could be the difference between a benign and a vulnerable program. We will not further elaborate on attention mechanisms, but the two papers cited in this paragraph are interesting reads.

Complementary and extended data sources

As discussed in our data generation chapter, an extension of our work would be to explore different code sources for training and testing data. This extension would, of course, require immense work with gathering and especially labeling. As we suggested, it could be possible to use a set of static and dynamic vulnerability detectors for labeling, but we do not possess the resources to carry out such tasks. Another interesting idea would be to create a dataset consisting of a combination of generated and scraped data samples. It would also be intriguing to increase the input vectors' complexity by introducing additional functions, more complex code structure, and recursive calls.

Classify multiple types of vulnerabilities

Even though buffer errors are the most ubiquitous type of vulnerability for the last 25 years [14], other vulnerabilities can be equally harmful. Race conditions, failures to validate input, are all vulnerabilities one would want to detect as early as possible. An exciting project would be the exploration of the ability to do multi-class classification between a set of different vulnerabilities. We acknowledge that such a task would most likely require a far more advanced neural network and probably some innovative feature engineering.

Pinpoint location of vulnerability

An impressive extension of our work would be if the neural network could predict where the problem is. This ability would be immensely useful in practice, especially with large code files. One idea is to have the RNN output a likelihood of vulnerability over each line of code, thus allowing the network to highlight where it is more likely to reside. It could also be possible to output likelihood over code chunks or functions. We argue that this extension would indeed be the most useful of our mentioned possibilities. It could be an invaluable tool for detecting and repairing vulnerable software.

Summary

This chapter has been a summary of our directions for the future of vulnerability detection by using deep learning techniques. We also provided an introduction to attention mechanisms.

Chapter 8

Conclusion

This last chapter summarizes our research goal and main findings deduced from our experiments. We consolidate our limitations, research results, and evaluation of experiments in order to establish a well-grounded conclusion. We close this thesis with some final thoughts and inspirations for professionals working in the field of research.

Our research aimed to explore the possibilities and limitations of vulnerability detection through Supervised Learning and Recurrent Neural Networks. In order to prove or disprove the hypothesis, we conducted a set of experiments on binary classification. Based on our qualitative and quantitative experiments, we can assess the usefulness of our novel approach.

Based on experiment 6.1, we observed some promising indications concerning learnable features in assembly language code. The experiment also validated our Neural Network design and implementation.

Experiment 6.2 underpins and proves that learnable features are indeed present in assembly code. By conducting this experiment, we are also able to observe that Recurrent Neural Networks exhibit capacity and potential for differentiating between two data classes. We can also conclude that our Neural Network is successful in detecting a single vulnerability located at a fixed position within a program file.

In experiment 6.3, we investigated performance and accuracy over a set of vulnerable functions. We significantly increased data complexity by allowing each negative sample to draw from a set of vulnerable functions. The results and observations reinforce and strengthen our conclusion about Recurrent Neural Networks' ability to detect vulnerabilities. We also observed a significant increase in the required computational efforts needed in order for our Neural Network to converge to a sufficient accuracy.

We designed experiment 6.4 intending to unveil our Neural Networks performance on low-variance data. We carefully designed a dataset where all positive and negative samples only consist of functions that use the same vulnerable C system-call. The benign, or positive samples, are repaired counterparts of the vulnerable samples. The results and observations indeed conclude that our approach exhibits high performance on low-variance data. We also observe the requirements for considerable resources when performing classification on exceedingly similar data. Even though our Neural Network trained on a small number of samples (a tenth of data in experiment 6.3), the required time to reach sufficient accuracy was almost identical.

Our last experiment, 6.5, further supports and advances previous findings. We conducted this experiment to establish a conclusion on whether or not our Neural Network can handle datasets of substantial size. We can draw several conclusions through the observations of this experiment. First, we observe that the Neural Network was successful in achieving an impressive accuracy over the data distribution. A related observation is that the Neural Network achieved this accuracy even though the number of generated samples for training and testing were a small subset of the full distribution. Again, we observe that, as the complexity of our experiments increases, so does the challenge in hyperparameter optimization.

Our results thoroughly indicate that there is a potential for vulnerability detection through the use of Recurrent Neural Networks. Our experiments have proven that Recurrent Neural Networks has excellent capabilities when it comes to extrapolating context in sequences of data. We have established that there are enough features in the assembly code language for a Neural Network to differentiate between two data classes. Through that conclusion, we can, at least with some certainty, state that there are some underlying similarities between natural language and programming languages. Thus, a positive statement for treating code as language follows.

Through our research, we can conclude that Recurrent Neural Networks can perform binary classification on a single type of vulnerability. Even with our shallow architecture, the approach holds excellent merit when it comes to predicting which programs are vulnerable and not. Our Network achieved high accuracy throughout our experiments. Even when resented with extremely low-variance data, the Neural Network was able to differentiate between classes successfully.

Even though our experiments were successful, we observe the increasingly difficult hyperparameter optimization when the Neural Network is exposed to increasingly complex datasets. It is indeed worth noting that the Neural Network is also highly sensitive to these configurations in the latter experiments. The increased sensitivity might indicate that our architecture is too

shallow, as it struggles to converge unless it is provided with "optimized" parameters.

We believe that through our research methodology and experiments, we have been able to establish a sufficient foundation of results to state our conclusion. Our experiments and approach are simplified and limited compared to the grand scheme of vulnerability detection. A self-generated dataset limits our ability to generalize our statement for all "real world" data. However, our methodology has provided us with unexpected insight into the potential of Recurrent Neural Networks' ability to perform vulnerability detection.

As we have conducted experiments with limited resources, further research is needed to determine the full potential of doing vulnerability detection on software with machine learning paradigms. However, we firmly believe that our research has sparked optimism towards further exploration and study of this field of research.

8.1 Final thoughts and inspirations

Applying machine learning paradigms to vulnerability detection is a challenging but interesting approach to information security. Our work has been intriguing, intense, and rewarding. We hope our work may be useful in furthering research. We encourage researchers to further explore the possibilities and limitations of vulnerability detection through machine learning paradigms.

Bibliography

[1] *Information technology — Security techniques — Information security management systems — Overview and vocabulary*. Standard. International Organization for Standardization, Jan. 2009.

[2] Various contributers. *WannaCry ransomware attack*. URL: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack. (accsessed: 09:06:2020).

[3] Sean Dillon and Dylan Davis. *MS17-010 EternalBlue SMB Remote Windows Kernel Pool Corruption*. URL: https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_eternalblue. (accessed: 21.03.2020).

[4] Various contributers. *The Shadow Brokers*. URL: https://en.wikipedia.org/wiki/The_Shadow_Brokers. (accsessed: 15:06:2020).

[5] The National Cybersecurity FFRDC. *Common Vulnerabilities and Exposures*. URL: https://www.cvedetails.com/browse-by-date.php. (accsessed: 09:06:2020).

[6] Bishop Fox. *Google Hacking Diggity Project*. Version 3.1. URL: https://resources.bishopfox.com/resources/tools/google-hacking-diggity/. (accsessed: 09:06:2020).

[7] Fabian Yamaguchi and the Joern team. *Joern A Robust Code Analysis Platform for C/C++*. Version 0.3.1. URL: http://mlsec.org/joern/docs.shtml. (accsessed: 09:06:2020).

[8] David A. Wheeler. *Flawfinder*. Version 2.0.11. URL: https://dwheeler.com/flawfinder/. (accsessed: 09:06:2020).

[9] 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. 8026–8037. URL: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] William Arild Dahl. *Vulnerability Detection usingRecurrent Neural Networks*. June 2020. URL: https://github.com/williamadahl/RNN-for-Vulnerability-Detection.

[11] OWASP Open Web Application Security Project. *Buffer Overflow Attack | OWASP*. URL: https://owasp.org/www-community/attacks/Buffer_overflow_attack. (accessed: 21.03.2020).

[12] James P. Anderson. "Computer Security Technology Planning Study." In: 2 (1972), pp. 60–61.

[13] Donn Seeley. "A Tour of the Worm." In: 1 (1988), pp. 8–9.

[14] Sourcefire Vulnerability Research Team (VRT). *Yves Younan*. URL: https://owasp.org/www-chapter-belgium/assets/2013/2013-03-05/OWASP_Belgium_Yves_Younan_2013.pdf. (accessed: 21.03.2020).

[15] Peter Vreugdenhil. *Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit*. URL: http://vreugdenhilresearch.nl/ms11-002-pwn2own-heap-overflow/#more-197. (accessed: 21.03.2020).

[16] Walter Pitts Warren S. McCulloch. "A logical calculus of the ideas immanent in nervous activity." In: *The bulletin of mathematical biophysics* 52.1 (1990), pp. 99–115. DOI: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[17] Various authors. *Neuron - Wikipedia*. URL: https://en.wikipedia.org/wiki/Neuron. (accessed: 22.03.2020.

[18] Jeffery A. Bilmes Shengjie Wang Tianyi Zhou. "Bias Also Matters: Bias Attribution for Deep Neural Network Explanation." In: *The 36th International Conference on Machine Learning* 1 (2019), pp. 1–3.

[19] Sagar Sharma. *Activation Functions in Neural Networks*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. (accessed: 23.03.2020).

[20] Andrej Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/neural-networks-1/. (accessed: 24.03.2020).

[21] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[22] Andrew Zisserman Karen Simonyan. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *conference paper at ICLR 2015* (2015), pp. 1–8.

[23] et al. Christian Szegedy Wei Liu. "Going deeper with convolutions." In: 1 (2014), pp. 1–10.

[24] Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun. "Deep Residual Learning for Image Recognition." In: 1 (2015), pp. 1–12.

[25] Ole-Johan Skrede. *Lecture notes in IN5400 - Machine Learning for Image Analysis*. Jan. 2019.

[26] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. (accessed: 21.05.2020).

[27] Jimmy Lei Ba Diederik P. Kingma. "Adam: A Method for Stochastic Optimization." In: *CoRR* abs/1412.6980 (2015).

[28] Sebastian Ruder. "An overview of gradient descent optimization algorithms." In: (2019).

[29] Andrej Karpathy et al. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/. (accessed: 21.05.2020).

[30] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[31] Ng Andrew. *Knuth: Computers and Typesetting*. URL: http://www.robotics.stanford.edu/~ang/courses.html. (accessed: 20.03.2020).

[32] "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: *Proc. IEEE* 105.12 (2017), pp. 2295–2329.

[33] Richard S. Sutton. *Lecture in Deconstructing Reinforcement Learning*. Aug. 2009.

[34] Kevin Gurney. *An introduction to neural networks*. Taylor & Francis, 1997. ISBN: 9781857285031.

[35] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/. 25.03.2020.

[36] Tollef Jahren. *Lecture notes in IN5400 - Machine Learning for Image Analysis*. Mar. 2019.

[37] "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches." In: (2018), p. 19.

[38] "An Empirical Exploration of Recurrent Network Architectures." In: (2015), pp. 6–8.

[39] "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models." In: (2018).

[40] "Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features." In: (2015).

[41] "A Comparative Assessment of Malware Classification using Binary Texture Analysis and Dynamic Analysis." In: *AISec '11: Proceedings of the 4th ACM workshop on Security and artificial intelligence* (2011), pp. 21–30.

[42] "Malware images: visualization and automatic classification." In: *VizSec'11: Proceedings of the 8th International Symposium on Visualization for Cyber Security* (2011), pp. 1–7.

[43] "Data Mining Methods for Detection of New Malicious Executables." In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001* (2000).

[44] "Software vulnerability prediction using text analysis techniques." In: *MetriSec '12: Proceedings of the 4th international workshop on Security measurements and metrics* (2012), pp. 7–10.

[45] "Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection." In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)* (2015).

[46] "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning." In: (2011).

[47] "Malware Detection by Eating a Whole EXE." In: (2017).

[48] "Learning the PE Header, Malware Detection with Minimal Domain Knowledge." In: *ACM AISec'17* (2017).

[49] "Automated Vulnerability Detection in Source Code Using Deep Representation Learning." In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018).

[50] "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection." In: *Network and Distributed System Security Symposium* (2018).

[51] "Collective Anomaly Detection based on Long Short Term Memory Recurrent Neural Network." In: (2017).

[52] "Deep Learning for Classification of Malware System Call Sequences." In: *AI 2016: AI 2016: Advances in Artificial Intelligence* (2016), pp. 137–149.

[53] "N-Grams-Based File Signatures For Malware Detection." In: *SCITEPRESS - Science and Technology Publications* (2009), pp. 317–320.

[54] "Vulnerability detection with deep learning." In: *2017 3rd IEEE International Conference on Computer and Communications (ICCC)* (2017).

[55] "Toward large-scale vulnerability discovery using Machine Learning." In: *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy* (2016), pp. 85–96.

[56] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer Science & Business, 2006. ISBN: 9780387310732.

[57] Microsoft Corporation. *An In-Depth Look into the Win32 Portable Executable File Format, Part 2.* URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/march/inside-windows-an-

in-depth-look-into-the-win32-portable-executable-file-format-part-2. (accsessed: 17:04:2020).
[58] "Learning Trees and Rules with Set-Valued Features." In: *AAAI-96 Proceeding* (1996).
[59] K-9 Mail. *k9mail*. https://github.com/k9mail/k-9. 2020.
[60] Microfocus. *Fortify Application Security*. URL: https://www.microfocus.com/en-us/solutions/application-security. (accsessed: 17:04:2020).
[61] Chris Wild. *The Wilcoxon Rank-Sum Test*. University of Auckland, Department of Statistics, 1997. URL: https://www.stat.auckland.ac.nz/~wild/ChanceEnc/Ch10.wilcoxon.pdf.
[62] National Instutute of Standards and Technology. *NVD-HOME*. URL: https://nvd.nist.gov/. (accsessed: 23:04:2020).
[63] National Instutute of Standards and Technology. *Software Assurance Reference Dataset Project*. URL: https://samate.nist.gov/SRD/index.php. (accsessed: 23:04:2020).
[64] Steven Cardinal. *Neptune.c – the Birth of SYN Flood Attacks*. SANS Institute, 2002. URL: https://pen-testing.sans.org/resources/papers/gcih/neptunec-birth-syn-flood-attacks-102303.
[65] Donald Bren Hall Irvine. *KDD Cup 1999 Data*. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. (accsessed: 23:04:2020).
[66] University of Oslo. *Abel computer cluster*. URL: https://www.uio.no/english/services/it/research/hpc/abel/. (accsessed: 14:06:2020).
[67] UNINETT Sigma2 AS. *Small scale exploratory work*. URL: https://www.sigma2.no/small-scale-exploratory-work. (accsessed: 02:06:2020).
[68] Alex Krizhevsky. *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/~kriz/cifar.html. (accsessed: 26.03.2020).
[69] Various contributors. *strcpy(3) - Linux man page*. URL: https://linux.die.net/man/3/strcpy. (accsessed: 26.03.2020).
[70] Various contributors. *strncpy(3) - Linux man page*. URL: https://linux.die.net/man/3/strncpy. (accsessed: 26.03.2020).

[71] Various contributors. *strcat(3) - Linux man page*. URL: https://linux.die.net/man/3/strcat. (accsessed: 26.03.2020).

[72] Various contributors. *scanf(3) - Linux man page*. URL: https://linux.die.net/man/3/scanf. (accsessed: 26.03.2020).

[73] Various contributors. *sprintf(3) - Linux man page*. URL: https://linux.die.net/man/3/sprintf. (accsessed: 26.03.2020).

[74] Various contributors. *gets(3) - Linux man page*. URL: https://linux.die.net/man/3/gets. (accsessed: 26.03.2020).

[75] Various contributors. *fgets(3) - Linux man page*. URL: https://linux.die.net/man/3/fgets. (accsessed: 26.03.2020).

[76] Various contributors. *memcpy(3) - Linux man page*. URL: https://linux.die.net/man/3/memcpy. (accsessed: 26.03.2020).

[77] Python Software Foundation. *itertools — Functions creating iterators for efficient looping*. URL: https://docs.python.org/3/library/itertools.html. (accsessed: 27:03:2020).

[78] Python Software Foundation. *random — Generate pseudo-random numbers*. URL: https://docs.python.org/3/library/random.html. (accsessed: 27:03:2020).

[79] Hex-Rays. *IDA Pro - Hex Rays*. URL: https://www.hex-rays.com/products/ida/. (accsessed: 29:03:2020).

[80] Binary Ninja. *Binary Ninja*. URL: https://binary.ninja/. (accsessed: 29:03:2020).

[81] The National Security Agency. *Ghidra*. URL: https://www.nsa.gov/resources/everyone/ghidra/. (accsessed: 29:03:2020).

[82] Tamar Christina GCC Wiki. *GCC Wiki*. URL: https://gcc.gnu.org/wiki. (accsessed: 29:03:2020).

[83] "DWARF Debugging Information Format V4." In: (2010), pp. 126–131.

[84] Tamar Christina GCC Wiki. *CFI support for GNU assembler (GAS)*. URL: http://www.logix.cz/michal/devel/gas-cfi/. (accsessed: 29:03:2020).

[85] Red Hat Enterprise. *AT&T Syntax versus Intel Syntax*. URL: http://web.mit.edu/rhel-doc/3/rhel-as-en-3/i386-syntax.html. (accsessed: 29:03:2020).

[86] GNU development team. *Using as Assembler Directives*. URL: https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html. (accessed: 30:03:2020).

[87] Texas Instruments. *Common Object File Format*. Texas Instruments Incorporated, 2009. URL: http://refspecs.linuxbase.org/elf/elf.pdf.

[88] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. TIS Committee. May 1995. URL: http://refspecs.linuxbase.org/elf/elf.pdf.

[89] *Control-flow Enforcement Technology Specification*. Intel Corporation. May 2019. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[90] NLTK Project. *nltk.tokenize package*. URL: https://www.nltk.org/api/nltk.tokenize.html. (accsessed: 22:05:2020).

[91] "Practical Recommendations for Gradient-Based Training of Deep Architectures." In: (2012).

[92] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[93] "Effective Approaches to Attention-based Neural Machine Translation." In: (2015).

[94] "Attention Is All You Need." In: *CoRR* abs/1706.03762 (2017).

List of Figures

2.1 Memory Map 20
2.2 Stack Frame 21
2.3 Safe usage of buf 22
2.4 Overwritten 22
2.5 Malicious Buffer Overflow 23
2.6 Simplified illustration of a neuron 25
2.7 Neuron represented as a simplified mathematical model 25
2.8 The sigmoid activation function 27
2.9 The tanh activation function 28
2.10 Simplified model of two Neural Networks 29
2.11 Forward and backward pass through a neural network 30
2.12 Binary Cross-Entropy loss 31
2.13 Gradient decent on parameter θk [25] 33
2.14 Dropout Neural Net Model [30]. 35
2.15 Deep learning in context of artificial intelligence. [32] 38
2.16 The basic structure of a cell in a RNN 42
2.17 Example RNN structures and their application. [35] 42
2.18 Diagram for Long Short Term Memory (LSTM). [37] 44
2.19 Diagram for Gated Recurrent Unit (GRU). [37] 46

5.1 Pipeline of data creation 95
5.2 Architecture stack 98

6.1 Single class classification, LR:1e−3. 110
6.2 Single vulnerability classification, LR:5e−4, BS: 80 112
6.3 Single vulnerability classification, LR:1e−3, BS: 80 113

6.4 Multiple vulnerability classification, LR:5e−4, BS: 80 115
6.5 Multiple vulnerability classification, LR:1e−3, BS: 80 116
6.6 Vulnerable and benign counterparts, LR:1.25e−4, BS: 10 . . . 118
6.7 Vulnerable and benign counterparts, LR:2.5e−4, BS: 10 119
6.8 Quantitative experiment, LR:5e−5, BS: 250 122
6.9 Quantitative experiment, LR:1e−4, BS: 250 123

Listings

2.1 Buffer overflow vulnerable example 22
5.1 Buffer overflow vulnerable data sample 72
5.2 A crashing function 72
5.3 Safe useage of strcpy 75
5.4 A vulnerable function from our dataset 75
5.5 Repaired version of listing 5.4 76
5.6 Arguments for our generator script 77
5.7 Sample argument stack for generating binaries 77
5.8 Function name in the compiled ASM format 79
5.9 Function name in the original C function 80
5.10 Uniform function pool from Cartesian product 81
5.11 Tuples created by Itertools 82
5.12 Compiler flags for our generator script 89
5.13 Prefix for a compiled binary sapmle 90
5.14 Generated assembler directives 91
5.15 Generated assembler directive references 91
5.16 .ident directive and .section directives in a binary 92
5.17 Tokenization of a line of code 92
5.18 Tokenization of a line of code after regex 93
5.19 Various length of each line of instructions 94
5.20 Dictionary mapping 99
5.21 Word to index mapping 99

5.22 Assigning index to each word 99
5.23 Input dimensions 100
5.24 Raw line of code 100
5.25 Raw code represented as numerical vector 100
5.26 Embedding in the forward phase 102
5.27 Embedding fed into the LSTM layers 102
5.28 Temporal dropout 102
5.29 Output from the LSTM layers is fed into the FC layer 103
5.30 Loss calculated by criterion (BCE) 103
6.1 Configurations. 109
6.2 Configurations. 111
6.3 Configurations. 114
6.4 Configurations. 117
6.5 Configurations. 121
A.1 Generated code sample. 148
A.2 Main function of A.1 after feature engineering. 149
A.3 Assembly code represented as a numerical vector 150

List of Equations

2.1 Sigmoid activation 26
2.2 Tanh activation 27
2.3 Activation of Node k in layer l 29
2.4 Cross entropy loss for a single sample 30
2.5 Binary Cross-Entropy loss for a single sample 32
2.6 Gradient decent 32
2.7 Supervised learning training 38
2.8 Supervised learning mapping 38
2.9 Supervised learning test 39

2.10 Unsupervised learning training 39
2.11 Reward calculation for an Agent in an Environment 40
2.12 Discount factor for each timestep in Reinforcement Learning . . . 40
2.13 RNN recurrence formula 41
2.14 Forward pass of a Vanilla RNN 43
2.15 The different gates in a LSTM 45
2.16 Input / output to a LSTM cell 45
2.17 The different gates in a LSTM 47
5.1 Possible permutations without repetition. 84
6.1 Example permutations. 107

Appendix A

Appendix

Below is an example of a generated program by us:

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

void yylcvpqcgadf();
void civhsivrxmhc();
//Vuln decl:
void oorhlzskiwfd(char* s, int n);

int main(int argc, char* argv[]){

    yylcvpqcgadf();
    civhsivrxmhc();
    //Vuln call:
    oorhlzskiwfd(argv[1], strtol(argv[2], NULL, 10));
    return 0;
}
void yylcvpqcgadf(char* s){
    char dest[15] = "Testing second";
    printf("Before: %s", dest);
    if(strlen(s) > sizeof(dest)){
        char longer[strlen(s)];
        strcpy(longer, s);
        printf("After: %s\n", longer);
    }else{
        strcpy(dest, s);
        printf("After: %s\n", dest);
    }
```

```
30 }
31 void civhsivrxmhc(){
32     int n;
33     printf("Enter number: \n");
34     scanf("%d", &n);
35     if(n>10){
36         printf("N is larger than 10!\n");
37     }else{
38         printf("N less than or equal to 10!\n");
39     }
40 }
41 //Vuln function:
42 void oorhlzskiwfd(char*s, int n){
43     char dest[n];
44     strncpy(dest, s, strlen(s));
45     printf("Copied string: %s\n", dest);
46 }
```

Listing A.1: Generated code sample.

Below is assembly code of the generated program above after our feature engineering is completed.

```
1  endbr64
2  push rbp
3  mov rbp , rsp
4  sub rsp , 16
5  mov DWORD PTR -4 [ rbp ] , edi
6  mov QWORD PTR -16 [ rbp ] , rsi
7  mov eax , 0
8  call yylcvpqcgadf
9  mov eax , 0
10 call civhsivrxmhc
11 mov rax , QWORD PTR -16 [ rbp ]
12 add rax , 16
13 mov rax , QWORD PTR [ rax ]
14 mov edx , 10
15 mov esi , 0
16 mov rdi , rax
17 call strtol @ PLT
```

```
18 mov edx , eax
19 mov rax , QWORD PTR −16 [ rbp ]
20 add rax , 8
21 mov rax , QWORD PTR [ rax ]
22 mov esi , edx
23 mov rdi , rax
24 call oorhlzskiwfd
25 mov eax , 0
26 leave
27 ret
```

Listing A.2: Main function of A.1 after feature engineering.

Line five of Assembly code from A.2 transformed into a numerical vector:

```
[3, 46, 8, 76, 6, 5, 7, 2, 77]
```

Listing A.3: Assembly code represented as a numerical vector

The numerical vector from A.3 transformed into features for our Recurrent Neural Network by the embedding layer:

$$\begin{bmatrix} [0.380, & 2.003, & -0.643, & 1.288, & 0.295, & 0.160, & -0.023, & 0.447, & -0.371] \\ [0.418, & -1.453, & -1.167, & 0.283, & -0.738, & 0.810, & 0.061, & 0.959, & 1.061] \\ [0.033, & -0.692, & -1.704, & 0.359, & 0.668, & 0.683, & 0.101, & -0.531, & -0.776] \\ [0.508, & 0.489, & -0.651, & 0.382, & 0.172, & -1.105, & 0.485, & 0.707, & 0.126] \\ [0.423, & 0.488, & -1.053, & -0.474, & 0.471, & -1.450, & -1.236, & -0.107, & -0.739] \\ [0.363, & 1.256, & 0.557, & 1.702, & -1.272, & -0.409, & -0.857, & 1.543, & 1.404] \\ [-0.115, & -0.451, & 0.406, & 1.498, & 0.873, & 0.285, & -0.193, & 0.572, & -0.999] \\ [0.656, & -1.017, & 0.087, & -0.717, & 0.901, & -0.457, & 2.186, & 1.035, & 1.712] \\ [-0.711, & 0.442, & 0.589, & -0.682, & -0.491, & 0.804, & -0.217, & -0.428, & 0.090] \end{bmatrix}$$