

**MASTER**

**Vulnerable code repair using Deep Learning**

Klarenbeek, Jordi R.

*Award date:*  
2021

[Link to publication](#)

**Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



Master Thesis

# Vulnerable code repair using Deep Learning

Author: Jordi R. Klarenbeek, 0904147

Study program: Information Security Technology

Graduation Supervisor: Vadim Liventsev

Version 1.2

Eindhoven, August 15, 2021

## Abstract

This work showcases multiple approaches to vulnerable code repair using deep learning. Deep learning has been proven to be effective in source code analysis tasks, such as vulnerable code repair. One of the existing problems is the lack of datasets containing complex vulnerabilities. The Big-Vul dataset contains 11823 vulnerable functions from 348 open-source projects in the C and C++ programming language and thus solves this problem. We explore how existing LSTM models, trained with this dataset, perform on the task of predicting the repaired code and the repair operations, such as add and delete. Our findings are that LSTM models that use the code's AST structure perform best. Furthermore, that the dataset is promising, but not large enough for a LSTM model to learn how to produce correct and functional programs. Pretraining the LSTM model with other C programs would be necessary for improving the output.

## Acronyms

<b>AST</b>	Abstract Syntax Tree
<b>ANTLR</b>	ANother Tool for Language Recognition
<b>BLEU</b>	Bilingual Evaluation Understudy, Papineni et al. (2002)
<b>CFG</b>	Context-Free Grammar
<b>CNN</b>	Convolutional Neural Network
<b>GP</b>	Genetic Programming
<b>GRU</b>	Gated Recurrent Unit, Cho et al. (2014)
<b>LSTM</b>	Long Short-term Memory, Hochreiter and Schmidhuber (1997)
<b>NLP</b>	Natural Language Processing
<b>NMT</b>	Neural Machine Translation
<b>RNN</b>	Recurrent Neural Network, Bengio et al. (1994)
<b>CVE</b>	Common Vulnerability and Exposures
<b>CWE</b>	Common Weakness Enumeration
<b>CWSS</b>	Common Weakness Scoring System
<b>CVSS</b>	Common Vulnerability Scoring System
<b>SLM</b>	Structural Language Model, Alon et al. (2020)

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Background</b>	<b>8</b>
2.1	Vulnerabilities and MITRE databases . . . . .	9
2.2	Encoder-decoder . . . . .	10
2.3	Recurrent Neural Network . . . . .	11
2.3.1	Long Term Short Memory (LSTM) networks . . . . .	11
2.3.2	Tree LSTM . . . . .	13
2.3.3	Gated Recurrent Unit (GRU) . . . . .	14
2.4	Attention Mechanism . . . . .	14
2.5	Vocabulary . . . . .	15
2.6	Edit models . . . . .	16
2.7	Vector embedding of language and code . . . . .	16
2.8	Metrics of translation effectiveness . . . . .	17
2.8.1	BLEU . . . . .	17
2.8.2	F-score . . . . .	17
2.8.3	Top-k accuracy . . . . .	17
2.8.4	Levenshtein distance . . . . .	18
<b>3</b>	<b>Related Work</b>	<b>19</b>
3.1	Models in literature . . . . .	19
3.2	Embedding layer setup . . . . .	21
3.3	Grammar correction models . . . . .	21
3.4	Research that uses Big-Vul Dataset . . . . .	22
<b>4</b>	<b>Models</b>	<b>23</b>
4.1	Implementation . . . . .	24
4.1.1	Software framework . . . . .	24
4.1.2	Sequential Encoder . . . . .	24
4.1.3	Tree Encoder . . . . .	24
4.1.4	Sequential Decoder . . . . .	24
4.1.5	Binary Classifier . . . . .	24
4.1.6	Implementation Testing . . . . .	24
<b>5</b>	<b>Dataset and Processing</b>	<b>25</b>
5.1	Preprocessing . . . . .	25

5.2	Sequential encoder data . . . . .	27
5.2.1	Padding . . . . .	27
5.3	Tree encoder data . . . . .	28
5.4	Edit decoder data . . . . .	28
5.5	Binary Classifier data . . . . .	28
5.6	Input and output sequence length . . . . .	29
<b>6</b>	<b>Method</b>	<b>31</b>
6.1	Hidden vector size . . . . .	31
6.2	Training . . . . .	31
6.3	Evaluation . . . . .	32
6.4	Teacher Forcing . . . . .	32
<b>7</b>	<b>Results</b>	<b>33</b>
7.1	Translation and Edit models . . . . .	33
7.2	Binary Classifier . . . . .	34
<b>8</b>	<b>Discussion</b>	<b>37</b>
<b>9</b>	<b>Conclusion</b>	<b>40</b>
	<b>Bibliography</b>	<b>42</b>

# Chapter 1

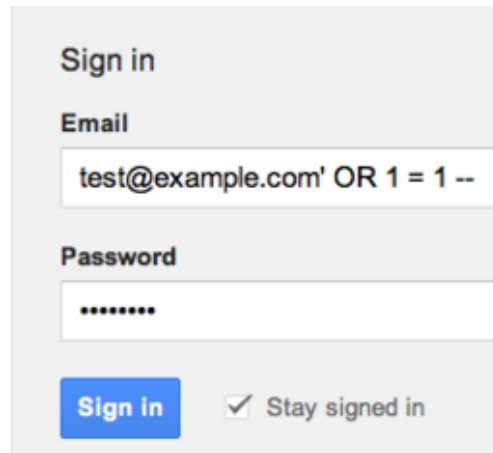
## Introduction

Repairing security vulnerabilities is an important part of the software development life cycle. Programs can contain millions of lines of code and can have many contributors. So, it is difficult to prevent vulnerabilities from existing. The potential damage of vulnerabilities has been proven by the ransomware attack by the REvil hacker group spread via the Kaseya IT management system (Kari, 2021). In this attack the Kaseya IT management software was compromised, resulting in malware being uploaded to their clients as an update to the software. The estimate is that 2000 companies have been hit by the ransomware and that the ransom for releasing all the data is 70 million dollars. Applying vulnerability repair before the implementation of software would decrease the risk of such an attack.

The problem with vulnerability repair is that the costs might outweigh the benefits. Solving vulnerabilities can be done by a human programmer, but this costs resources and is not fool proof. Human programmers have created the vulnerability and thus there is no insurance that the repair will not create a new vulnerability. Another approach used in the past was using genetic programming (Monperrus, 2018), in genetic programming a few operations are chosen which mutate the source code. A test suite or oracle is used to quantify the success of the mutation. The most successful mutation would be used to further mutate the program. In the case of using genetic programming, an oracle needs to be constructed that can validate the new mutations, for example using a compiler that gives errors with faulty code or a test suite that tests for the desired outcomes. Genetic programming has the downside that the oracle and test suite needs to be defined by a human, so it is also prone to human error. An example where genetic programming is useful is for shortening the runtime of a program, as it is easy to compare two mutations on their runtime. Comparing two mutations on security requires us to define what secure code is and that is difficult.

The difficulty with defining vulnerability and defining rules for capturing vulnerable software is due to the diverseness and fluidity of the concept. The word is often applied when a threat actor can misuse a system or access data that they were not supposed to. Vulnerable is not something that software is by itself, software is vulnerable because a specific action or series of actions makes it vulnerable. For a classic example of this we can look at an SQL injection. In such an injection the attacker inputs SQL code into a HTML web form used to input

username and password. Example is given in figure 1.1. The HTTP form should only get text and no special characters, but if an end string token is passed (') as regular text, then the parser that parses the SQL request will have a problem since it will read this (') token as part of the programming language. That causes everything after this token to not be processed as text, but as SQL. This can be abused by adding SQL commands after this token, allowing for unauthorized access of the database or logging in with the admin account by inserting "pass' OR 1=1 --", which would be always be equivalent to true even as "pass" is not the admin's password, as 1=1 is always true. Important to note is that the HTML form and the SQL parser are not doing something wrong separately, but how they communicate together is what creates the vulnerability. This makes it difficult to define the vulnerability.



The image shows a web form titled "Sign in". It has two input fields: "Email" and "Password". The "Email" field contains the text "test@example.com' OR 1 = 1 --". The "Password" field contains "\*\*\*\*\*". Below the fields are two buttons: a blue "Sign in" button and a checkbox labeled "Stay signed in".

Figure 1.1: Basic SQL injection

Finding vulnerabilities and fixing vulnerabilities is a never ending process, but smart solutions can be created to help defenders stay ahead of attackers. A possible solution would be to use deep learning, to let a neural network learn the definition of a vulnerability and work with this definition. Deep learning has been shown to work well in tasks where the rules behind certain decisions or categories are fuzzy and hard to define, for example with the cat or dog classification task in computer vision (Liu et al., 2014).

There has been research on using deep learning for repairing code. Examples are the SequenceR model of Chen et al. (2019) and the Deepfix model of Gupta et al. (2017). The Deepfix model was able to repair common programming errors, such as a missing semicolon or a missing curly bracket. The SequenceR model was able to produce one-line patches. The problem with current research is that the vulnerabilities repaired are simple and thus would easily be spotted by a competent programmer. As such, these repair models would not be useful for complex vulnerabilities that have evaded the detection by competent programmers and static analysis tools.

Therefore there is need for models that research automatic repair of complex vulnerabilities. To help with this the Big-Vul C/C++ code vulnerability dataset was created by Fan et al. (2020). The dataset is unique in the high number of real-world vulnerabilities and the richness of information available for each vulnerability. This dataset has been constructed by scraping GitHub for C/C++ vulnerabilities and matching fixes. All the vulnerabilities



are registered in the public Common Vulnerabilities and Exposures (CVE) database. The dataset contains 3754 vulnerabilities of 91 different vulnerability types, all accompanied with a solution to the vulnerability. In the literature there has not been many deep learning models that use the Big-Vul dataset. So, exploring how different deep learning models train and function with this dataset will be valuable for the scientific community. The research question this thesis will try to answer is: What is the best design for a deep learning C/C++ vulnerability repair model trained with the Big-Vul dataset?

First section after the introduction will be the background information that will be useful in understanding the method, models and metrics. Next section will be an analysis of earlier work on the subject of source code analysis. The related works section will be followed up with the model section, this section will relate how the models were created. The method section contains the steps that were taken to perform the experiments. The dataset section explains what the dataset contains and the steps that were taken to preprocess the data. The results section will show the results for the implemented models. Lastly the conclusion will discuss the findings and the implications of those findings.

# Chapter 2

## Background

In the literature there exists a lot of research on vulnerable software repair with deep learning. Allamanis et al. (2018) talked about the naturalness hypothesis of code, that programming languages are a form of communication and have similar statistical properties to natural languages. This would mean that natural language processing models can be applied to programming language analysis. An important distinction between programming languages and natural languages is that the structure of code is more fixed than for natural languages. A compiler needs to be able to interpret the code and a compiler is not as flexible as a human. This shows the bimodality of code, it needs to be read by a human and by a computer. Another important distinction is that programming languages are defined by a few designers at big software companies, while natural languages change all the time from the bottom-up.

The naturalness hypothesis might be extended to draw a comparison between grammatical errors in natural language and vulnerabilities in programming languages. Deep learning models have been able to learn the statistical properties of grammatical errors (Zhao et al., 2019), so they might also be able to learn the properties of vulnerabilities. A prerequisite for our vulnerable code repair model would be that there are learnable statistical properties between vulnerable code and non-vulnerable code in the dataset. The absence of these statistical properties would mean that a neural network will have problems with learning to translate vulnerable code into non-vulnerable code, as there is no learnable mapping from vulnerable to non-vulnerable.

Natural language analysis is often based on sequential analysis methods. N-grams were the first attempts at analysing natural language, which looked at a corpus of data to determine which words are most probable in the company of other words. Next the RNN networks were used, such as GRU and LSTM (Hochreiter and Schmidhuber, 1997), which had the benefit from n-grams to process input in a particular order. In the last four years a new sequential solution has been proposed and used, the transformer (Vaswani et al., 2017). This transformer uses multi-headed attention mechanisms to dynamically process a sequence, so not purely from left to right. Secondly a Transformer can map longer dependencies in the data. Downside of the transformer network is that it requires vast quantities of data to train

properly.

Analysing code with deep learning can be solved with similar sequential models as for natural language. The only big difference is that programming languages have a strong functional structure and that it is useful for the analysis and transformation of code to process this structure. When a compiler compiles a program, it first parses the program into an Abstract Syntax Tree (AST). This AST is used to generate the machine code needed to execute the program on the Central Processing Unit (CPU). This AST structure can be processed in multiple forms, the tree paths can be embedded instead of individual tokens or the AST can be sequentialized in a way that retains information from the AST structure. Another way is to influence the embedding of a token by the parent node's embedding or the child node's embedding. The most straightforward approach would be to use a model that is structured as a tree and uses the token embeddings, using a tree-LSTM for example.

The task of vulnerable code repair can be compared with two natural language processing tasks. The first is language translation and the other is grammar correction. In language translation the network takes a sequence of tokens in the input language, for example French, and translates it into the output language, for example German. For our task the input language would be insecure code and the output language would be secure code. Translating the insecure code into secure code would fix the vulnerabilities. Positive about this approach is that the network will be able to correct vulnerabilities that are very complex and span multiple lines, since it completely rewrites the program. Negative is that this approach will need a lot of data, as it is difficult to have a decoder produce long sequences.

The second approach is to approach the task as grammar correction. This would require the network to learn to produce the edit operations that are needed to fix the input sequence. These edit operations would be predefined and the model will only need to learn when to apply these edits. Positive about this approach is that less data should be needed, since shorter sequences have to be produced. Negative is that preprocessing is necessary to create the train and test output dataset and the repair operations chosen will limit the transformative capacity of the neural model. As the model will only be able to produce the operations.

## 2.1 Vulnerabilities and MITRE databases

To increase the public knowledge on security vulnerabilities, the Common Vulnerabilities and Exposures (CVE) database was created by MITRE (<https://cve.mitre.org/cve/>). This database contains many of the vulnerabilities that are discovered by security researchers. The goal is to lift the secrecy around vulnerabilities and not to leave this knowledge only on black markets. Each vulnerability is given a unique ID starting with CVE, then the year of occurrence and then some random numbers. The vulnerability used in the EternalBlue hacking tool, which allowed access to Microsoft for the Wannacry ransomware attack and the NotPetya cyberattack, is classified as CVE-2017-0144.

The CVE's are scored on their severity with the Common Vulnerability Scoring System (CVSS). the scoring is separated in Base metrics and Impact metrics. The Base metric measures how a vulnerability would be accessed, for example only on the local network, and

the complexity of that access. The Impact metric measures the impact on confidentiality, Integrity and Availability. There have been multiple ways of calculating the score. In the current version 3 of the CVSS, the EternalBlue vulnerability was scored an 8.1 by the National Vulnerability Database of the USA.

As described in the introduction it is difficult to define what makes code vulnerable. Still it is possible to a certain extent to categorize vulnerabilities and improve the communication between security researchers and developers. For this the Common Weakness Enumeration (CWE) system was developed. Every vulnerability that is disclosed in the Common Vulnerability and Exposures database is categorized with a CWE. The EternalBlue vulnerability was classified as CWE-20: Improper Input Validation. Other existing categories are cross-site scripting (CWE-79), Out-of-bound Write (CWE-787) and SQL injection (CWE-89).

The CWE's are also scored on their severity with the Common Weakness Scoring System (CWSS). This score is an indicator for which vulnerabilities would need to be prioritized for repairing. The metric is divided in three metric subgroups: Base Finding, Attack Surface and Environmental. Figure 2.1 shows an overview of the metric subgroups. The Base Finding quantifies the inherent risk of the weakness. The Attack Surface quantifies the difficulties an attacker needs to overcome to use the weakness. Environmental quantifies how the environment impacts the risk of the weakness. These subgroups are combined in a score between 0 and 100 indication the risk of a CWE. For example a SQL injection vulnerability has a CWSS of 20.69 and a Cross-site Scripting vulnerability has a score of 46.82.

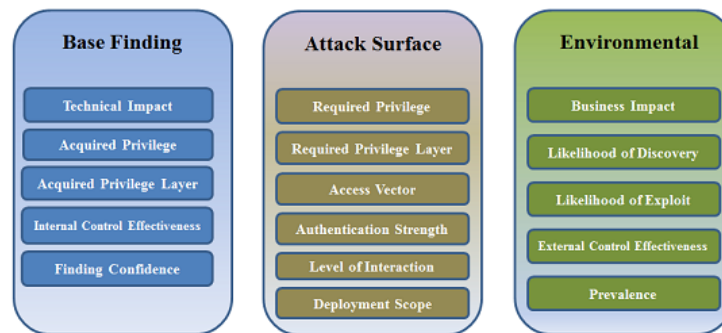


Figure 2.1: CWSS metric groups

## 2.2 Encoder-decoder

Transforming data to another form is often done in deep learning with an encoder-decoder architecture (Cho et al., 2014). An encoder-decoder pair contains an encoder model and a decoder model. The encoder encodes the input to a context vector that represents the input. The context vectors of all inputs form a vector space where closeness of vectors can be interpreted as inputs being similar. The decoder takes the context vector and reconstructs it according to how the decoder is trained. The encoder-decoder architecture is very flexible and can be used for many tasks. For example for the task of translating a sentence from one language to another language or for image caption generation, where the image processing encoder creates the context vector and a language processing decoder reconstructs the vector

into a sentence. In our case we will use language processing models for both encoder and decoder.

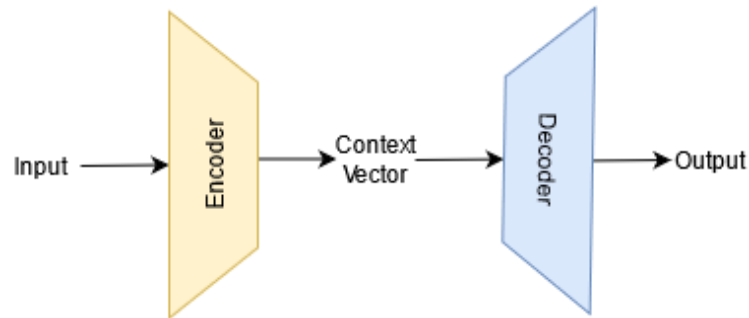


Figure 2.2: Encoder-Decoder architecture

## 2.3 Recurrent Neural Network

The neural network that is most relevant for our research is the Recurrent Neural Network (RNN). Fully connected networks are not good at processing sequences, as they cannot take into account the order of the tokens in a sequence. The Recurrent Neural Network (RNN) was designed for the purpose of processing sequences (Bengio et al., 1994). A RNN network processes the order of input by having cells influence each other.

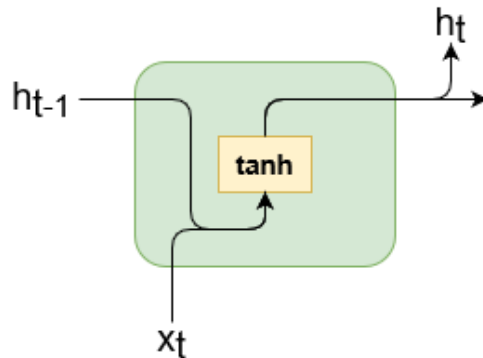


Figure 2.3: RNN cell

A RNN cell produces a hidden state, which it passes to the cell processing the next token in the sequence. A sequence of 5 token would use 5 cells, with the weights matrix being the same for each neuron. The hidden state is also the output of the model. An example is shown in Figure 2.3. The value of  $y_0$  is equal to  $h_1$ . Figure 2.4 shows how multiple RNN cells would be chained in sequence.

### 2.3.1 Long Term Short Memory (LSTM) networks

A RNN works good for sequences, but there is a problem with vanishing or exploding gradients for long sequences. To deal with this the Long Short-Term Memory network was developed (Hochreiter and Schmidhuber, 1997). With the LSTM model the cell contains a cell state  $c$  that can be updated by a cell and influences the outgoing hidden state. This allows the network to retain knowledge over longer sequences.

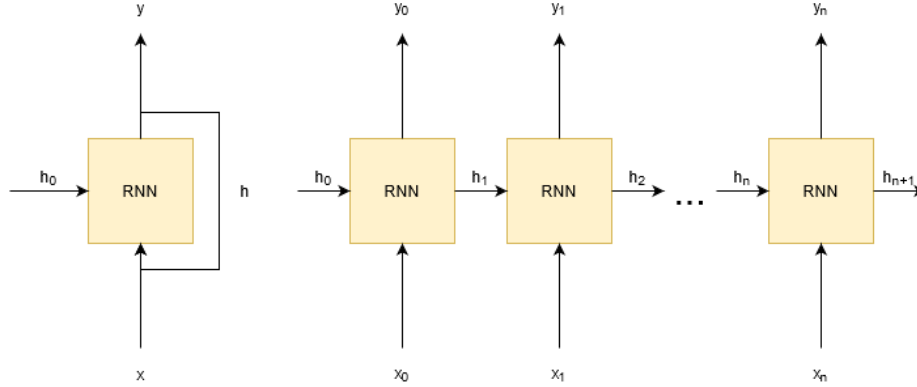


Figure 2.4: Recurrent Neural Network

The LSTM cell is divided in three parts: the forget gate, the input gate and the output gate.

The forget gate allows the cell to remove any knowledge of the previous cell state or to pass it on. The input gate processes the input from the previous hidden state and the next input in the sequence. The output gate combines the calculated cell state and the output state to calculate the hidden state and the output of the cell. Each of the gates are calculated by separate formula.

For the forget gate formula,  $f_{n+1}$  is the output of the forget formula for position  $n + 1$  in the sequence.  $x_{n+1}$  is the input token embedding for position  $n + 1$  in the sequence.  $U$  is the first weight matrix for the forget gate, which is multiplied with the input token embedding.  $h_n$  is the hidden state of the previous LSTM cell of position  $n$  in the sequence.  $W$  is the second weight matrix for the forget gate. For the input gate and output gate the formulas are the same, except with different weight matrices  $U$  and  $W$  for each gate. The  $\sigma$  stands for the sigmoid activation function.

The forget gate:

$$f_{n+1} = \sigma(x_{n+1}U^f + h_nW^f)$$

The input gate:

$$i_{n+1} = \sigma(x_{n+1}U^i + h_nW^i)$$

The output gate:

$$o_{n+1} = \sigma(x_{n+1}U^o + h_nW^o)$$

For calculating the input activation vector  $\bar{C}_{n+1}$  the hyperbolic tangent is taken from the input  $x_{n+1}$  multiplied with the weight matrix  $U$  plus the previous hidden state  $h_n$  multiplied with the weight matrix  $W$ .

$$\bar{C}_{n+1} = \tanh(x_{n+1}U^g + h_nW^g)$$

With the input activation vector  $\bar{C}_{n+1}$  we can calculate the new cell state. The output of the forget gate is elementwise multiplied with the previous cell state, this allows the model to "forget" the previous cell state. The input gate  $i_{n+1}$  is elementwise multiplied with the

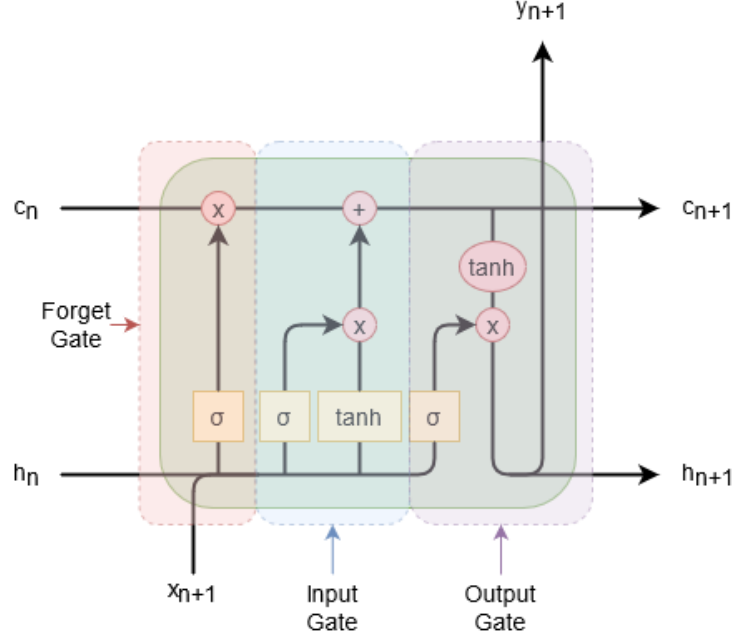


Figure 2.5: LSTM cell

input activation vector  $\bar{C}_{n+1}$ . The sum of both Hadamard products is put through a sigmoid function.

$$C_{n+1} = \sigma(f_{n+1} \odot C_n + i_{n+1} \odot \bar{C}_{n+1})$$

The new hidden state  $h_{n+1}$  is calculated by multiplying the output gate  $o_{n+1}$  with the hyperbolic tangent of the new cell state  $C_{n+1}$ .

$$h_{n+1} = \tanh(C_{n+1}) \odot o_{n+1}$$

### 2.3.2 Tree LSTM

The LSTM network is very good at processing sequential data and learning sequential patterns. In the case of programming language analysis it would be better to process the tree structure of the data. Tai et al. (2015) created for this purpose a tree structured encoder LSTM model with two variants. The child-sum variant and the N-ary variant.

In the child-sum variant the children hidden states are added into a combined hidden state  $h_j$  which is processed similar to a normal LSTM. In the formula below  $C(j)$  is the set of children of node  $j$ .

$$h_j = \sum_{k \in C(j)} h_k$$

The second difference with normal LSTM is that there is a forget gate for each child. So a parent cell has the choice to ignore its left child's cell state and not its right child's cell state. In the formula below  $C(j)$  is again the set of children of node  $j$ .

$$C_j = i_j \odot \bar{C}_{n+1} + \sum_{k \in C(j)} f_{jk} \odot c_k$$

With the N-ary variant each child gets its own parameter matrix. The formula below shows how the matrix child multiplications are combined in formula for calculating the input gate  $i_j$ . Each child gets its own  $U$  matrix.  $N$  is the number of children. This formula would calculate the input gate for node  $j$ .

$$i_j = \sigma(W^{(i)}x_j + \sum_{l=1}^N U_l^{(i)}h_{jl} + b^{(i)})$$

The child-sum variant is best for trees with a high branching factor or where the children are unordered. The branching factor is the amount of child nodes. The N-ary variant is most useful when there is a recurrent pattern with the locations of tokens, for example if the left child always contain a noun phrase and the right child a verb phrase in natural text analysis. The N-ary variant is also best with a low branching factor. For our situation the child-sum variant is best, as the AST's of the different programs are very different and as such have a high branching factor.

### 2.3.3 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) is a form of RNN cell (Cho et al., 2014), designed to limit the effect of exploding and vanishing gradients. Figure 2.6 shows the structure of the cell. The GRU cell contains a forget gate, which allows the model to remember the hidden state. In many ways it is similar to the LSTM, but since it only has a hidden state it has less parameters.

## 2.4 Attention Mechanism

A drawback of the encoder-decoder setup is that the decoder only receives info about the input via the context vector and thus is not very flexible. The context vector that is passed between the encoder and decoder, gives a good summary of the input sequence, but it might contain more information about the latter half of the sequence and less on the first half. To solve this problem the attention mechanism was introduced. The attention mechanism works similar to how humans focus on a particular word while reading a sentence. It allows the decoder to focus on parts of the encoder hidden states. Concretely the attention mechanism is a matrix that is multiplied with the input sequence, the result is added in the decoder model.



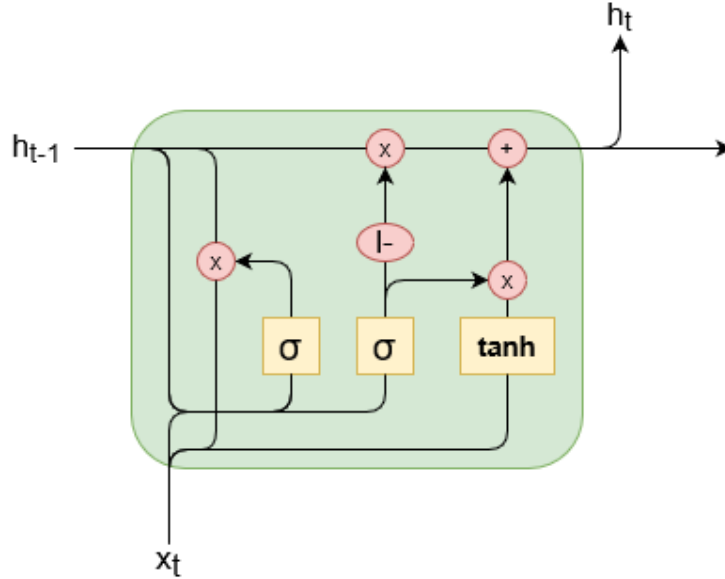


Figure 2.6: GRU cell

## 2.5 Vocabulary

Similar to natural languages, programming language have some words that are used more often than others. For example "int" will occur more often than a function defined in an obscure library. Furthermore since programmers can define variable and functions names, there are almost infinite possible tokens. To use our data we need to determine a fixed vocabulary, as this will be needed to define the size of the embedding space. So choices need to be made on the size of the vocabulary, a large vocabulary might allow our model to process a complex input of tokens, which would allow the model to better process certain information of a program. Negative about a large vocabulary is that more data is needed to correctly learn the embedding space. Another problem is that some tokens do not occur often in a text, so the model has little context to learn the embedding vector of these rare tokens. A small vocabulary would be easier to learn and it would simplify which choices the decoder model needs to make, but it might also lead to the input becoming generic and thus not giving the model useful information to learn.

There are a few possibilities for decreasing the vocabulary size, all centered around handling the infinite number of programmer defined tokens. A possible approach could be to use  $\langle \text{UNK} \rangle$  (unknown) tokens, these tokens will be used as placeholder for function and variable names. Downside of this approach is that it might result in a very small vocabulary. Upside is that it is easy to implement. Another approach could be to use a copying mechanism, which is used to replace any  $\langle \text{UNK} \rangle$  tokens in the output predicted by the decoder with tokens from the input where  $\langle \text{UNK} \rangle$  was not yet inserted. A possible third approach could be to separate the user generated tokens in multiple categories: variable names and function names. These categories can each get a token, for example  $\langle \text{VAR} \rangle$  and  $\langle \text{FUNC\_NAME} \rangle$ . This will allow the model to learn the place of these different categories in the embedding space. A twist on this approach would be to number these tokens, so for example if there are 3

variable names in a function, these are tokenized as  $\langle \text{VAR1} \rangle$ ,  $\langle \text{VAR2} \rangle$  and  $\langle \text{VAR3} \rangle$ . This would help with compiling the output programs as a compiler needs unique and consistent variable names. If all variables are represented with the same  $\langle \text{VAR} \rangle$  token, then the "same" variable will be defined multiple times and cannot be correctly parsed by the compiler.

## 2.6 Edit models

As shortly discussed at the start of this chapter, there are two main approaches for our model. A translation model and an edit model. An edit model would focus on predicting the edit operations needed to fix the buggy code and less on transforming the full code fragment. An edit model needs to predict two types of data, the edit operations and the new tokens. A model might need to decide to add or delete a line. In the case of an add, the model would also need to decide which tokens need to be added. Predicting these two types of data might be achieved by using two separate models, one for the tokens and one for the operations. Another approach would be to add the edit operations as tokens to the body of text. For example encapsulating an added line of code with special tokens  $\langle \text{add} \rangle$  and  $\langle / \text{add} \rangle$ , this is easy to implement and reduces the complexity of the model.

## 2.7 Vector embedding of language and code

Besides the set-up of the neural network, another important part of the processing is how the word embeddings are created. A neural network needs numerical input and as such it cannot accept a word as input. For training our network we need to create a numerical representation of the words in our vocabulary. With natural language processing often a vocabulary is created with all the unique existing words. The position of each word has a position in this vocabulary. In the example below a vocabulary is constructed of the words "I", "am", "a", "neural" and "network". Each word is given a position in the vocabulary, this can be used to translate a sentence with these words in to numbers. The sentence "a neural network I am", would be translated into  $\{3, 4, 5, 1, 2\}$ .

$$\text{Vocabulary} = \{1 : "I", 2 : "am", 3 : "a", 4 : "neural", 5 : "network"\}$$

These positional numbers do not give any insight in the relation between words. For example 'queen' and 'king' are close as words semantically, but this could not be reflected in their position in the vocabulary. A single number cannot hold enough information for a model to learn the semantics of a word. A solution is to express a word with a vector. In natural and programming language models a word embedding vector is a vector that is used to represents a word. The place of a word in the embedding vector space says something about the properties of a word. 'queen' and 'king' should be close to each other in the embedding space.

In a RNN network a linear embedding layer is added, to serve as a lookup table for the embedding vector of a word. This embedding layer is a fully connected layer with as output the embedding vector. The embedding layer is trained together with the whole network, as such the embedding space that is constructed in the training phase cannot be interpreted.

## 2.8 Metrics of translation effectiveness

The task the model will need to perform is similar to natural language translation. Except in this case the bad code needs to be translated into good code. Since it is similar, we can use metrics used in natural language processing to measure how well a machine translates, also for this task.

### 2.8.1 BLEU

The BLEU score, BiLingual Evaluation Understudy, is a method of quantifying the quality of a machine translation (Papineni et al., 2002). It can be used for evaluating natural language processing and programming language processing. It combines the modified n-gram precision with a correction based on the length difference of machine translation and desired translation. The n-gram precision is the amount of n-grams match between the machine translated sentence and the real translation. An n-gram is a subpart of a sentence, for example the sentence "I like to eat cookies", can be divided into 5 1-grams: {"I", "like", "to", "eat", "cookies"}, 4 2-grams: {"I like", "like to", "to eat", "eat cookies"}, 3 3-grams: {"I like to", "like to eat", "to eat cookies"}, 2 4-grams: {"I like to eat", "like to eat cookies"} and into 1 5-gram {"I like to eat cookies"}. If a machine learning algorithm would output "I like eat only brownies". The Bleu score would be calculated with the amount of same n-grams. So 3/5 1-grams, 1/4 2-grams and none of the other n-grams.

### 2.8.2 F-score

The F-score is a measure for the accuracy of predictions, It is the harmonic mean of the precision and recall. Precision is the percentage of true positives in all the positive classifications. Recall is the percentage of true positives in all datapoints that should have been classified as positive. For an example we could look at a model that detects cancer. The precision of this model would be the amount of correctly diagnosed patients, compared to the total amount of patients diagnosed by the model. The recall of this model would be the amount of correctly diagnosed patients, compared to the total amount of patients that have cancer.

$$F_1 = \frac{tp}{tp + \frac{1}{2}(fp + fn)}$$

Tp is the number of true positives, the number of correct positive classifications. Fp is the number of false positives, the number of incorrect positive classifications. Fn is the number of false negatives, the number of incorrect negative classifications. The F-score is not relevant for the encoder-decoder models, as for these model the BLEU score will be more informative to the success of the model. For the binary classifier the F-score could be informative.

### 2.8.3 Top-k accuracy

Top-k accuracy is a fairly straightforward measure and is best used with beam search, where the model predicts multiple candidates. The k indicates where the correct answers can be and still be counted as a correct prediction. For example with top-5 accuracy, if the model's fourth candidate is the target output that would count as a win for the model, with top-3

accuracy this would not be a win. With top-k accuracy it does not matter if the fourth candidate or first candidate is correct, it is only important if the correct candidate is in the top-k candidates.

#### **2.8.4 Levenshtein distance**

The Levenshtein distance is the minimum number of single character changes to a sequence to turn it into another sequence. This edit distance can be used as a metric for quantifying how different two words or two sequences are. Often this distance is calculated with the separate characters in a word. In the case of source code analysis it might be applied on a token level.

The word error rate is derived from the Levenshtein distance and works on the word level and not the character level.

# Chapter 3

## Related Work

### 3.1 Models in literature

The history of automatic software repair is fairly brief, but nevertheless rich (Monperrus, 2018). In the 90s genetic programming started gaining popularity as a method of automatic program synthesis, Koza (1992) described genetic programming in detail in his book "Genetic Programming". Wotawa and Stumptner (1996) outlined an approach for model-based software debugging using a set of test cases. In the late 2000s approaches were developed for using genetic programming to repair programs, examples of this were the approach of Forrest et al. (2009) and of Murali et al. (2009). A practical implementation of these approaches was the Genprog model (Le Goues et al., 2011), which used the compiler as an oracle for choosing the mutations that fix a program.

After improvements in natural language processing, source code analysis also began to use deep learning models. Similar to natural language processing sequential oriented techniques are mostly used. Although there have been some models that use Convolutional Neural Networks (CNN), Russell et al. (2018) used a CNN model with a kernel the width of the token embedding. The task of their model was automatic vulnerability detection. Downside of their model is that it cannot map long dependencies without losing efficiency, as increasing the height of the kernel to map a longer dependency also means more neuron weights. Another CNN model was made by Mou et al. (2016), their model used a 1D convolutional layer for classifying the function of code.

Due to the exploding and vanishing gradient problem vanilla RNN is not suitable for analysis of long sequences. The Gated Recurrent Unit (GRU) model was developed to overcome this problem, the Deepfix model of Gupta et al. (2017) used GRU in their RNN for fixing common programming errors. Similar to Genprog, they used the compiler as an oracle to decide whether Deepfix needed to continue changing the code.

Another RNN variant that is commonly used is the LSTM (Hochreiter and Schmidhuber, 1997), which can encode very long sequences. Examples of using LSTM for source code processing are the SequenceR model of Chen et al. (2019), the CODIT model of Chakraborty et al. (2020) and the Structural Language Model (SLM) of Alon et al. (2020). These three

models operate with the same network setup, the difference being how they embed the input data. The CODIT model translated the AST's into Context Free Grammar (CFG) rules and processed those in sequence, the SequenceR model embeds the tokens without using the AST structural information, Alon et al. encode the path from all leaf to the place where code needs to be predicted.

Besides seq2seq models there also exist models that process the input as trees, so in that case there is no need to preprocess the AST into a sequence. Tai et al. (2015) designed such a tree-LSTM. In their paper they used the model for sentiment classification. Tai et al. used the tree LSTM for encoding the input, another difficulty is to generate tree structured output with a tree lstm decoder. Alvarez-Melis and Jaakkola (2016) developed an approach in which the topology of a tree is first generated and next the tokens of the nodes were produced using a combination of the hidden states of the parents and of the siblings, combining knowledge of the parents and the siblings of a node.

Most of the RNN models above have a similar method of learning, they output the probability of a particular token and use categorical cross-entropy to quantify how wrong the predicted output is. Another approach was used by Harer et al. (2018), they used a GAN setup with a RNN generator and CNN discriminator. Instead of discriminating between real and fake generated output, the discriminator needed to determine what secure code was and what insecure code. The generator was pretrained as a de-noising autoencoder, to ensure that it has a good starting point.

In 2017 the transformer network was proposed by Vaswani et al. (2017), this transformer network has taken an increasingly prominent place in sequential data analysis. The Transformer uses no RNN model, but a multi-headed attention mechanisms to dynamically link sequence tokens to each other. Attention mechanisms were earlier applied in the RNN models, but not as the primary way of processing data. Another important finding of Vaswani et al. was the use of a positional encoding, this means adding some value to the token vector embedding to indicate the corresponding token's place in the sequence. This is needed as the embedded tokens are put into a fully connected layer and as such without the positional encoding the network would not be able to learn location of tokens in the sequence.

The transformer model has shown great promise for natural language processing. Since a transformer model needs a lot of data to train, many approaches use pre-training and fine-tuning. For pre-training often unsupervised data is used and after pre-training the network is fine-tuned on a particular task with supervised data. Examples of this is the GPT2 model (Radford et al., 2019) and the BERT model (Devlin et al., 2018). Based on the BERT model, Feng et al. (2020) created the codeBERT model, which uses the structure of the BERT model for bimodal source code analysis tasks. Bimodal in the sense that the tasks involve transforming programming language into natural language, for example code comment creation. Another interesting source code transformer is the tree-transformer of Harer et al. (2019). This network uses a Tree Convolutional block to process the AST tree and does not use any positional encoding. Researchers at Facebook (Kim et al., 2020) used a pre-trained GPT2 model to do next token prediction. The most interesting of their paper is their use and comparison of different embedding strategies, which will be further highlighted in the next section.

## 3.2 Embedding layer setup

Processing code into a set of embedding vectors is a topic which is very crucial for the functioning of our model. The embedding layer will need to embed the input code in a way that most information is stored about the input tokens and program. One crucial part of code is its inherent structure, the structure that a compiler uses to translate the high-level language into assembly language. As earlier stated the AST structure is the representation created by the compiler of the structure of a program and using this AST will benefit our model greatly. Focussing on the AST structure also makes the solution space smaller as the solutions are constrained by the grammar rules of the AST (McKay et al., 2010). There have also been models that use the program in sequence with no regard for the AST structure, codeBERT and SequenceR for example. A program is for these models only sequence of tokens with no structure.

Other approaches embed the structural information of the AST into the token embedding or embed paths instead of tokens. The SLM model (Alon et al., 2020) uses the paths to the unknown node from all leaf nodes and root. Every path is encoded with a LSTM model and the resulting encodings are aggregated. Another approach by Kim et al. (2020), sequentialized the AST by embedding leaf nodes together with their respective root path, encoding the rootpath with a LSTM. The leaf embeddings were entered into the transformer without positional embedding. Kim et al. also used a depth-first sequentialization that did not embed the paths but only in what order the nodes were embedded. Harer et al. (2019), as mentioned earlier, used a tree convolutional block. This TCB layer uses the embedding of the parent node and left sibling nodes to influence the embedding of a node. This TCB layer is also used instead of the feedforward layers in the encoder and decoder.

There have also been attempts at developing a positional encoding that will help the transformer learn the position of a token in the tree. Shiv and Quirk (2018) proposed a possible tree related positional encoding limited to binary trees. Their paper was not published since the depth and breadth of their evaluation was found lacking. So it is an interesting idea, but not yet fully developed.

## 3.3 Grammar correction models

The task of code repair can be seen as a translation task or a grammar correction task. Most of the models mentioned above use the translation approach. Chen et al. (2021) use the BigVul dataset to predict the correct changes to buggy code and as such they approach it as a grammar correction task. There also exist examples in literature for using the editorial approach for natural language grammar correction, which is quite similar with the only difference that natural language is corrected and not a programming language.

Awasthi et al. (2019) created a grammar correction model called the Parallel Iterative Edit model, it outputs edits instead of sequences and it uses a parallel inference method instead of beam search. The PIE model predicts edits for all the tokens in parallel, so the edits do not influence each other. They model iteratively applies predicts edits until the sequence is correct. The PIE model used a pretrained BERT model to encode the sequences. The edit operations were add, delete, replace and clear (no edit), the edits are seen as labels placed

on the original edit.

Another grammar correction model by Zhao et al. (2019), featured a copy-augmented architecture. Their model was a seq2seq model pretrained as a denoising auto-encoder. Zhao et al. handled the grammar correction task as a translation task, translating the incorrect sentence into a correct sentence. Zhao et al. use a copy mechanism to handle the out of vocabulary tokens.

### **3.4 Research that uses Big-Vul Dataset**

In the literature there are 4 papers that cite the Big-Vul dataset. In the paper of Xu et al. (2020), they referenced to the dataset in a paper analysing the memory related CVE's on the Rust programming language. The reference was used to compare the amount of executable bugs in Rust with the amount in C/C++. Li et al. (2021) used the dataset to train their vulnerability detection model IVDetect, together with two other C/C++ datasets from Chakraborty et al. (2021) and Zhou et al. (2019). The programs in the dataset were represented as program dependence graphs and graph-based classification was applied to find vulnerable code. The paper got a mean average precision increase of 27% over the baseline of 50%. The third paper was the master thesis published by Michl (2021). His master thesis was an analysis of faulty software design which can lead to software vulnerabilities and possible artificial intelligent solutions to mitigate them. He uses the dataset to analyse vulnerabilities in C/C++. The last paper of Chen et al. (2021), uses the dataset to train a neural network to repair the security vulnerabilities in the dataset. They create a custom bug-fix dataset by scraping Github and pretrain their VRepair transformer model with this data. The Big-Vul dataset is used to finetune the VRepair transformer model. The programs were inputted as sequences and no method was used to encode the AST structure of the programs. The VRepair model achieved 17,3% accuracy in predicting the fixes for the vulnerabilities.



# Chapter 4

## Models

In the background and related works section multiple possible networks were showcased. The overarching model architecture for the translation and edit models will be Encoder-Decoder, as this has been shown to be most effective in similar tasks, for example for the SequenceR model and the Deepfix model. The next choice that needs to be made is what network the encoder will be and what network the decoder will be. Considering the literature there are three main networks we can choose: the sequential LSTM, the tree LSTM and the transformer network. The transformer network requires a lot of data, more than is available in the Big-Vul dataset. A possible solution would be to manually scrape Github for more bug-fix data and pretrain a transformer with this information. Although this would cost much time to do and this has already been tested by Chen et al. (2021). What they did not research is the effectiveness of using a neural network that is specifically designed for tree inputs. So it will be interesting to compare the sequential encoder LSTM with the tree encoder LSTM. Due to time constraints only the sequential decoder will be explored and not the tree LSTM decoder. For the predicted output we have two possible outputs, the sequentialised parse trees and the edit patches. The naming scheme of the models will be enc2dec, indicating to what the model transforms the data. For a tree encoder the model will start with tree and with a sequential encoder it will start with seq. A model with tree encoder and seq decoder will be a tree2seq model. An encoder-decoder model that predicts the edits will end with edit, so for example seq2edit.

To evaluate the existence of statistical properties between vulnerable and secure code, an encoder LSTM architecture will be used to encode the information of the programs in the dataset. Multiple fully connected layers ending in a single neuron will decide on the vulnerability of the code. If this binary classifier architecture will be able to distinguish between vulnerable and non-vulnerable code, then that would be an argument in favor of statistical learnable difference between vulnerable and non-vulnerable code in our dataset.

In total we have 4 encoder-decoder models: seq2seq, tree2seq, seq2edit and tree2edit. We have two binary classifiers: seq2clas and tree2clas.

## 4.1 Implementation

### 4.1.1 Software framework

For constructing the deep learning models the Python programming language was used, with the Pytorch machine learning library. Pytorch allows for extensive customization of the model implementation, which is essential as source code analysis is a fairly niche area of deep learning and there is no deep learning library that contain prefab models for this purpose. As source code analysis is closely related to natural language processing we can use natural language models as inspiration for our models.

### 4.1.2 Sequential Encoder

The sequential encoder is a fairly straightforward encoder model. It contains an embedding layer and an LSTM layer. The LSTM has two layers and is bidirectional.

### 4.1.3 Tree Encoder

The tree encoder is based on the pytorch-tree-lstm repository (Dawe, 2019). This tree LSTM implementation is based on the child-sum design of Tai et al. (2015). It uses the node order of a tree, the order in which nodes in a tree need to be processed, to determine which node needs to be passed through the LSTM layer. It 'batches' LSTM inputs based on the order in which they need to be processed. This approach also allows us to batch trees, which greatly decreases training time. Other approaches to the tree LSTM use recursion to get to all the nodes and to process them bottom up, but this is slow and costs a lot of memory. Also with recursion it is not possible to batch trees.

### 4.1.4 Sequential Decoder

The decoder receives the hidden and cell state of the seq encoder or tree encoder and uses it to initialize the first cell, after initializing, again with a for loop the decoder LSTM is fed the hidden and cell state of the previous iteration. For the teacher forcing the target tensor tokens are used, without teacher forcing the predictions of the model is used. The LSTM has two layers and is unidirectional.

### 4.1.5 Binary Classifier

The binary classifier uses the hidden vector and passes it through 2 fully connected layers, with the last layer being passed through a sigmoid layer into a single neuron.

### 4.1.6 Implementation Testing

The implementation of each model was checked by overfitting on a small portion of the dataset. To achieve this overfitting, the models were trained on 10 datapoints for 300 epochs and also tested on these same 10 datapoints. This showed that the models are able to learn and that the metrics are functioning correctly.

# Chapter 5

## Dataset and Processing

The Big-Vul dataset (Fan et al., 2020) contains only vulnerabilities that are linked to the CVE database, as such all vulnerabilities have a CVE identification and a CWE classification. The vulnerabilities are from the period 2002 until 2019. In total there are different 91 CWE's in the dataset. The three most often occurring CWE types are Improper Restriction of Operations within the Bounds (CWE-119), Improper Input Validation (CWE-20) and Out-of-bounds Read (CWE-125).

The columns in the dataset are shown in Table 5.1. From these columns we mostly use the `func_before` and `func_after` since this contains the vulnerable functions and fixes. The columns `CWE_ID` and `CVE_ID` are used to analyse the effect of the models on different types of vulnerabilities. In preprocessing the `vul` column is used to split the functions into vulnerable functions and non-vulnerable functions. If this column is set to 0, the `func_before` is the same as the `func_after`.

### 5.1 Preprocessing

The dataset gathered by Fan et al. (2020) contains a lot of data, but it is not ready to be used by a deep learning network without preprocessing. For the networks we will need the programs' parse tree as a sequence or the tree itself in some format, for example as a json tree or Python dictionary.

First step in the preprocessing pipeline is to split the MSR dataset into non vulnerable functions, the vulnerable functions and the fixed functions. At this stage the code, program id, CWE ID, CVE ID and Vulnerability Classification is saved, the other data is not passed along as it is not needed for our network or any analysis.

The second step in the preprocessing pipeline is to parse the programs into Abstract Syntax trees. On a first attempt the Clang compiler was used, which has an option for intercepting the AST's before they are passed to the compiler. This resulted in only 66% being parsed without error, so around 6000 bug-fix pairs. The low amount was caused by the dataset containing code fragments and not full programs. Fan et al. (2020) collected vulnerable functions and divided a full program into fragments to separate these vulnerable functions.

Table 5.1: Columns of Big-Vul dataset

<b>Features</b>	<b>Description</b>
Access Complexity	Reflects the complexity of the attack required to exploit the software feature misuse vulnerability
Authentication Required	If authentication is required to exploit the vulnerability
Availability Impact	Measures the potential impact to availability of a successfully exploited misuse vulnerability
Commit ID	Commit ID in code repository, indicating a mini-version
Commit Message	Commit message from developer
Confidentiality Impact	Measures the potential impact on confidentiality of a successfully exploited misuse vulnerability
CWE ID	Common Weakness Enumeration ID
CVE ID	Common Vulnerabilities and Exposures ID
CVE Page	CVE Details web page link for that CVE
CVE summary	CVE summary information
CVSS Score	The relative severity of software flaw vulnerabilities
Files Changed	All the changed files and corresponding patches
Integrity Impact	Measures the potential impact to integrity of a successfully exploited misuse vulnerability
Mini-version After Fix	Mini-version ID after the fix
Mini-version Before Fix	Mini-version ID before the fix
Programming Language	Project programming language
Project	Project name
Publish Date	Publish date of the CVE
Reference Link	Reference link in the CVE page
Update date	Update date of the CVE
Vulnerability Classification	Vulnerability type
func_before	The function before the vulnerability being fixed
func_after	The function after the vulnerability being fixed
lines_before	The modified lines in the function before the vulnerability being fixed
lines_after	The modified lines in the function after the vulnerability being fixed
vul	"1" means vulnerable function and "0" means non-vulnerable function
vul_func_with_fix	The code comments showing how the vulnerability was fixed

The code fragments miss information that the Clang parser needs to successfully parse the functions.

The second attempt involved ANTLR. Which is a text parser developed for language analysis. The trees that are generated by ANTLR are not used for compiling code, but only for language analysis. Therefore the parser does not care about the code being fragments of programs. The parser only cares about whether the text matches the grammar file. For simplicity the C grammar was used, as 98% of the dataset contains C functions and the C grammar is less complex than the C++ grammar. To prevent problems with parsing C++ programs with the C grammar, all the C++ functions were dropped from the dataset.

Negative about the ANTLR C grammar is that it ineffectively handles expressions. Compound expressions such as " $var1 || var2 \rightarrow var3$ " or " $var1 * var2 = var3 \&\& var4$ " are difficult to handle, since they contain so many expressions. The solution of the C grammar was to check the existence of all possible expressions and save the existing expressions. This would create long tree branches of practically useless nodes, which would only make it more difficult to train a model on this data. To solve this specific problem and in general to reduce the size of the parsed tree, all nodes are skipped that have only one child. This ensures that the neural models only train on nodes that are relevant for the structure of the program.

Another difficulty that arose with parsing were how C macro definitions were used as placeholders for zero or more variables. An example of such a definition is `function(x, y, z DEF_DEF)`, where `DEF_DEF` could be used to inject `", a, b"` or `", a"` into the function call. The definitions could also be replaced with an empty string. As the C grammar did not anticipate these definitions, it could not correctly match them. To solve this a Regex pattern was used to replace the definitions with an empty string.

The ANTLR grammar still has some matching errors after the above mentioned fixes, the ANTLR program places `<ERROR>` tokens in the tree at these locations and continues parsing. My assumption is that there is enough structure left in a tree for the model to learn from and that it is still useful to use the AST even if there was a parsing error. The possible downside could be that the produced output is not usable as parts are missing in the predicted fix.

## 5.2 Sequential encoder data

After the parse trees are created they need to be made into sequences for the sequential encoder. To achieve this the parse trees were travelled with the depth-first algorithm. Figure 5.1 shows the order in which this particular tree would be traversed. This order is used to create a sequence out of the parse tree. The sequences were started with a `<START>` token and the sequences were ended with an end of function token `<EOF>`.

### 5.2.1 Padding

The sequential encoder uses a fixed length input to process the input sequences. For example if the limit is set to 500 tokens, then all sequences need to be of length 500 to process these sequences in batches. The sequences that were shorter than this limit were padded with zeros during the preparation of the data.

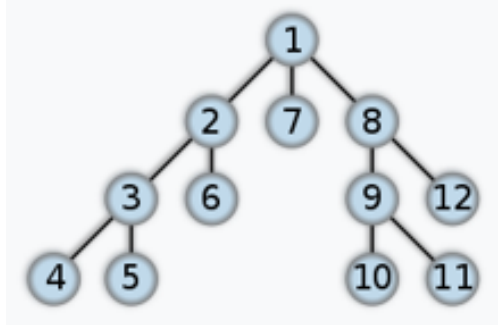


Figure 5.1: Depth-first traversal

### 5.3 Tree encoder data

The implementation of the tree encoder used is from the Github `pytorch-tree-lstm` repository (Dawe, 2019). The data needed for using this software needs to be a different format than a tree datastructure. The implementation needs a list with node features, an `adjacency_list`, `node_order` and `edge_order`. The list with node features contains the number paired with the token in the vocabulary. The `adjacency_list` contains which node are connected with each other. The `node_order` contains in which order the nodes need to be processed by the LSTM. The root node will get the highest number in this list and the leaf nodes will get a 0, as they can be processed at the first calculation step. The `edge_order` is similar, but then handling when an edge needs to be taken into account. This edge order combined with the adjacency list allows the implementation to process the child-parent information. During the preprocessing phase the trees are processed into these dataformats.

### 5.4 Edit decoder data

For the `seq2edit` and `tree2edit` model the Github patches are used from the Big-Vul dataset. These patches contain the information about which changes occurred on the vulnerable functions to fix the vulnerability. Extra tokens are added to indicate the start of a patch (`<patch>`) and the lines that are added begin with `<add>` token and the lines that are deleted begin with `<del>`.

### 5.5 Binary Classifier data

The input data used for the binary classifier was the vulnerable functions and the repaired functions. So, the model had to classify the vulnerabilities and the repaired code of the vulnerabilities. The output was either the function is vulnerable "1" or non-vulnerable "0" for the repaired functions. In total the dataset for the binary classifiers with sequential LSTM contained 17339 datapoints, with a maximum length of 1000 on the input sequences. The dataset for the tree LSTM contained 21272 datapoints, with no upper bound on length or depth.

```

@@ -1643,7 +1656,7 @@ static int samldb_check_user_account_control_acl(struct samldb_ctx *ac,
     return ldb_module_operr(ac->module);
 }

-     ret = dsdb_get_sd_from_ldb_message(ldb_module_get_ctx(ac->module),
+     ret = dsdb_get_sd_from_ldb_message(ldb,
                                         ac, res->msgs[0], &domain_sd);

     if (ret != LDB_SUCCESS) {

```

Figure 5.2: Github commit

```

<patch> static int samldb_check_user_account_control_acl(struct samldb_ctx *ac,
     return ldb_module_operr(ac->module);
 }

<del>ret = dsdb_get_sd_from_ldb_message(ldb_module_get_ctx(ac->module),</del>
<add>ret = dsdb_get_sd_from_ldb_message(ldb,</add>
                                         ac, res->msgs[0], &domain_sd);

     if (ret != LDB_SUCCESS) {

```

Figure 5.3: Edit patch

Table 5.2: Table with lengths and amount of sequences

Length	AST Sequence output	Edit output
200 tokens	3199	1091
500 tokens	6511	3859
700 tokens	7632	5164
1000 tokens	8602	6440

## 5.6 Input and output sequence length

The programs in the Big-Vul dataset can be very long and this might be a problem for the LSTM network. Predicting a sequence that is of length 100000 is more difficult than a sequence of length 10, as the model will need to 'remember' data longer. Therefore we will need to chose the maximum length of the input and output sequences. If the length is too short, there are less datapoints to train the model on. The longest program in the dataset is 61639 tokens long, the average length in the dataset is 790 tokens. Table 5.2 shows the number of programs that fit a particular maximum length. All the sequences that have maximum length 200 are also included with the sequences that have maximum length 500. This table was made with the length of the output sequences. During the experiments the models will be trained on these 4 different maximum sequence lengths. Figure 5.4 shows the distribution of sequence lengths below maximum length 5000. In Table 5.2 it is visible that the edit sequences are often longer than the output sequences. This indicates that the patches are long and span the whole function.

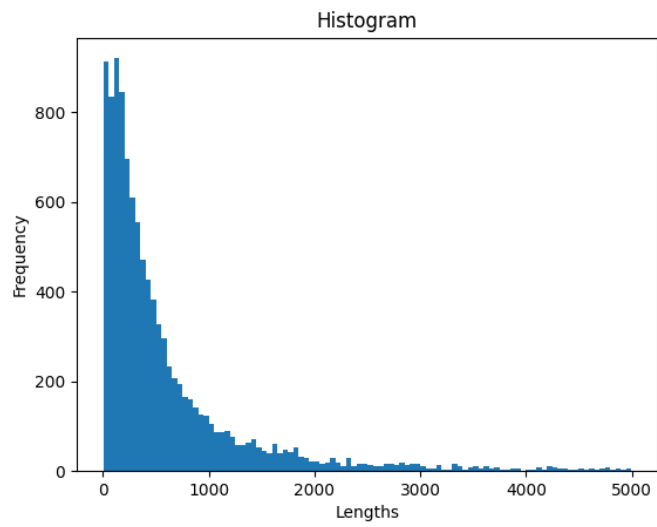


Figure 5.4: Distribution of program length with max length 5000



# Chapter 6

## Method

For our experiments there are some differences in how the models work, but there are also some similarities. The similarities are described in this section. The differences are described in chapter 4.

### 6.1 Hidden vector size

The hidden vector size will determine how much information can be stored in the hidden vector and thus can be memorized by the LSTM network. The hidden vector size will also determine how many parameters the model contains and thus the memory use and training time of the model. So the trade-off is between information stored in the model and training time.

### 6.2 Training

The models were trained on the train dataset for multiple epochs. 90% of the dataset was used to train the model. The other 10% were used to test the model. Early stopping was used to stop the training phase if the difference between the current loss was less than 1% of the average loss of the last 5 epochs. Minimum amount of epochs was 25, to ensure that the model trained for some time before stopping. The Adam optimizer was used to train the models with a starting learning rate of 0.01. For the loss function categorical cross-entropy was used for the translation and edit models. The binary classifiers used the binary cross-entropy loss function.

As demonstrated in the introduction it is difficult to define security vulnerabilities and insecurity. Ideal would be to give feedback to the model on the quality of a repair and to use a loss function that quantifies how secure the produced output is. A possible approach could be a discriminator that decides whether a function is vulnerable, similar to a discriminator in a Generative Adversarial Network (Goodfellow et al., 2020). This approach would require more data as the data used to train the generator cannot be used to pretrain the discriminator. For our models the choice was made to compare the produced output with the fixes available in the dataset. This approach ignores the possibility of other repairs that could be made on a vulnerability, but this is mitigated by the diversity of vulnerabilities and repairs

in the dataset. Since the vulnerabilities are fixed by different programmers, the assumption is that this teaches the model different ways of repairing the vulnerable functions.

## 6.3 Evaluation

In the evaluation phase the models are tested on their effectiveness with the BLEU score and with top1 accuracy. The BLEU score measures the similarity and as such is useful in determining how close the produced output is to the desired output. The evaluation function only compares the strings up until the end of function token or the beginning of the zero padding. It is possible that a model predicts the zero padding correct, but not the EOF token. The top1 accuracy is useful in determining how many exact matches are produced by the model.

## 6.4 Teacher Forcing

Another important part of training our network will be how we are going to correct the wrong output. The goal is for the network to learn how to fix bad code into good code. Vulnerable code can be repaired in multiple ways, the challenge is how the network will learn this. The approach mostly used in literature is to use Categorical Cross-Entropy for the produced output with the expected output. With this method the network will learn only one fix per vulnerable program, but to a certain extent this is mitigated by the data coming from different programmers that solved bugs in different ways. So to a certain extent this approach learns the model the statistical properties of fixing bugs.

Besides the loss function, it is also important to consider if wrong output should also be inputted into the decoder for predicting the next token in the tree. Another approach would be forced learning, in this approach the input of the decoder is not the previous output of the decoder, but the tokens of the expected outcome.

# Chapter 7

## Results

In this section the results are shown of the experiments. In section 7.1 we will show the results for the translation and edit models. In section 7.2 we will show the results for the binary classifier.

### 7.1 Translation and Edit models

Table 7.1 shows the BLEU scores for the different lengths of input and output sequences. All models have a hidden size of 512 and learning rate of 0.01. For the models with a tree encoder, the lengths requirement is set on the length of the output. The values in the table are an average of multiple training sessions. The base line column is what the BLEU score would be if the vulnerable function is compared with the repair, so if no change is made to the input function.

Table 7.1: BLEU scores of translation and edit models

	Seq2Seq	Seq2Edit	Tree2Seq	Tree2Edit	Base line
200 tokens	0.0826	0.0847	0.4205	0.0341	0.3358
500 tokens	0.0577	0.0409	0.4930	0.0222	0.2896
700 tokens	0.0617	0.0242	0.607	0.0153	0.2429
1000 tokens	0.0138	0.0177	0.0124	0.0000	0.1735

Below an example output is given of a produced program. Most of the produced information is abstract parse tree tokens. The <UNK> tokens represent variable and function names. The <ERROR> token represents parts of the program that the parser failed to parse. A deparser based on the ANTLR C grammar would be needed to deparse it to a program.

```
[ '<START>', 'translationUnit', 'externalDeclaration', 'externalDeclaration', '<ERROR>',  
'externalDeclaration', 'declarator', '<UNK>', '<ERROR>', '<ERROR>', 'externalDecla-  
ration', 'declarator', 'directDeclarator', '<UNK>', '(', 'parameterTypeList', 'parameterDec-  
laration', 'declarationSpecifiers']
```

## 7.2 Binary Classifier

The binary classifier had the task to classify whether a function was vulnerable or not. The dataset for training contains 50% vulnerable and 50% non-vulnerable (repaired functions). Therefore the baseline accuracy would be 50%, as this would be as good as random guessing by the model. Table 7.2 contains the results of our model with different hidden sizes, these hidden sizes are given in the first column.

Table 7.2: Accuracy scores of binary classifier

	Seq	Tree
256	50.75%	50.99%
512	50.70%	49.90%
768	48.70%	49.88%

Table 7.3 contains the results per CWE for the binary classifier with the seq encoder. The model was trained 3 times and the sum number of test cases per CWE is noted in column "Total Number", column "Correct Guesses" contains the number of times the model correctly guessed whether a fragment of code with this CWE was vulnerable or non-vulnerable. The CWE types were retained for the non-vulnerable fragments to check how successful the model was in classifying similar fragments of code. Since the test cases in the test dataset were randomly selected, not everytime the same CWE's were classified. So multiple training sessions were done to get results on more CWE types. The sum of column "Total Number" is the number of test cases in the three training instances. Table 7.4 shows similar results per CWE but for the binary classifier with the tree encoder.

Table 7.3: Accuracy per CWE for the seq classifier

<b>CWE</b>	<b>Correct Guesses</b>	<b>Total Number</b>	<b>Accuracy</b>
CWE-404	11	19	0,579
CWE-264	171	298	0,574
CWE-617	18	32	0,563
CWE-79	17	31	0,548
CWE-284	74	139	0,532
CWE-285	17	32	0,531
CWE-19	19	36	0,528
CWE-59	11	21	0,524
CWE-732	41	79	0,519
CWE-119	603	1167	0,517
CWE-369	31	60	0,517
CWE-20	208	406	0,512
CWE-125	153	301	0,508
CWE-200	101	204	0,495
CWE-000	385	781	0,493
CWE-476	65	132	0,492
CWE-254	27	55	0,491
CWE-787	52	108	0,481
CWE-835	13	27	0,481
CWE-415	12	25	0,480
CWE-190	61	128	0,477
CWE-494	15	32	0,469
CWE-189	52	111	0,468
CWE-416	59	126	0,468
CWE-17	22	47	0,468
CWE-399	163	353	0,462
CWE-362	70	156	0,449
CWE-400	34	78	0,436
CWE-704	24	60	0,400
CWE-94	6	18	0,333
CWE-310	3	12	0,250

Table 7.4: Accuracy per CWE for the tree classifier

<b>CWE</b>	<b>Correct Guesses</b>	<b>Total Number</b>	<b>Accuracy</b>
CWE-369	5	8	0,625
CWE-358	7	12	0,583
CWE-94	11	19	0,579
CWE-787	97	170	0,571
CWE-347	18	32	0,563
CWE-362	66	121	0,545
CWE-17	35	65	0,538
CWE-79	40	75	0,533
CWE-134	17	32	0,531
CWE-119	666	1258	0,529
CWE-269	38	72	0,528
CWE-190	82	156	0,526
CWE-284	63	120	0,525
CWE-310	35	67	0,522
CWE-416	88	172	0,512
CWE-189	151	298	0,507
CWE-20	342	676	0,506
CWE-476	71	143	0,497
CWE-125	115	234	0,491
CWE-000	596	1214	0,491
CWE-254	57	117	0,487
CWE-264	90	185	0,486
CWE-404	45	93	0,484
CWE-399	219	453	0,483
CWE-664	10	21	0,476
CWE-362	15	32	0,469
CWE-79	15	32	0,469
CWE-77	13	28	0,464
CWE-200	153	334	0,458
CWE-400	10	22	0,455
CWE-59	23	58	0,397
CWE-22	12	32	0,375
CWE-617	11	32	0,344
CWE-285	1	3	0,333
CWE-426	2	13	0,154
CWE-354	0	1	0,000

# Chapter 8

## Discussion

During the experiments two different encoders were compared, namely the tree LSTM and the sequential LSTM. The sequential LSTM used a sequentialized version of the function's parse tree. The results show that the tree encoder was superior to the sequential encoder in processing the input data, in the case of the translation models (seq2seq vs tree2seq). For the edit models the tree encoder performed worse. In the case of the seq2edit and tree2edit, the decoder does not predict in any way the parse structure of the function, as such it might be disruptive to use an encoder that is focused on the parse tree structure.

Comparing the results of the edit models with the translation models, it is visible that the translation models worked better than their respective edit counterpart. The difference is smaller between seq2seq and seq2edit than between tree2seq and tree2edit. As mentioned earlier using a tree encoder might be negative for predicting the edits. As the edits contain both the lines that are removed and are added, there is a high amount of duplicate code. This duplicate code might disrupt the tree structure of the program. Another reason could be that the predicted tokens do not contain parse tree information, as the github patches of the edit output are not parsed by ANTLR. Therefore the model might have difficulty predicting the edits, as there is no tree structure to learn.

Besides comparing the 4 different models with each other, it is also important to compare them with the baseline results. The baseline is to directly calculate the BLEU score similarity between the vulnerable functions and the fixes in the dataset. Table 7.1 shows that the baseline scored better than most models. Except for the tree2seq model, which scored higher on sequence lengths 200, 500 and 700. Interesting is that the scores of tree2seq increased even with the length of the sequence increasing, except for length 1000. For the other models the scores decreased as the length increased. For the tree2seq model the amount of datapoints might have had a stronger effect on training the model, then for the other models. It seems that at length 1000 this effect decreased strongly and this disrupted the ability of the model to learn to produce correct repairs.

The programs produced by the best model tree2seq show how the model produced the parse tree structure of a function. The translation models produce a sequentialized parse tree of the repair. An example can be seen in the produced function below. This repair was made

on a CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor) vulnerability. The BLEU score between the predicted repair and the actual repair was 0.39. As can be seen the produced repair correctly guesses the actual repair until the closing bracket token, after which all tokens were wrong. This problem occurred in multiple programs produced by the tree2seq model. The model has problem recovering from a wrong guessed token, probably since all further predicted tokens are based on this wrong guess. This issue might be solved by adding a self-attention mechanism to the decoder. A self-attention mechanism would allow the decoder to focus on previous predicted words and not only on the cell state and hidden state of the previous LSTM cell.

**Predicted repair:** [`<START>`, `translationUnit`, `externalDeclaration`, `externalDeclaration`, `<ERROR>`, `externalDeclaration`, `declarator`, `<UNK>`, `<ERROR>`, `<ERROR>`, `externalDeclaration`, `declarator`, `directDeclarator`, `<UNK>`, `(`, `parameterTypeList`, `parameterDeclaration`, `declarationSpecifiers`, `<UNK>`, `declarator`]

**Actual repair:** [`<START>`, `translationUnit`, `externalDeclaration`, `externalDeclaration`, `<ERROR>`, `externalDeclaration`, `declarator`, `<UNK>`, `<ERROR>`, `<ERROR>`, `externalDeclaration`, `declarator`, `directDeclarator`, `<UNK>`, `(`, `)`, `compoundStatement`, `,`, `blockItemList`, `expression`, `primaryExpression`, `<UNK>`, `(`, `)`, `->`, `<UNK>`, `(`, `argumentExpressionList`, `<UNK>`, `)`, `;`, `,`, `<EOF>`]

In the paper of Chen et al. (2021), they propose a vulnerability repair model using the transformer network and training with the Big-Vul dataset. Their VRepair model got a 17,3% accuracy on predicting repairs. Their approach is similar to our edit model. Focussing on the part of the code fragment that needed to be repaired, the difference is that they predict only the added and removed tokens instead of whole lines. The TFix model by Berabi et al. (2021), used a transformer model pretrained on natural language and achieved 46.3% in predicting exact matches. Both models VRepair and TFix outperformed our model, as our model achieved 0% top-1 accuracy. This is most likely attributable to the length of the sequences our model needs to predict, increasing the difficulty of producing an exact match. The other models only needed to predict small parts of a program.

The results of the binary classifier show that the neural network could not learn the difference between the vulnerable functions and non-vulnerable functions. The results are not significantly different from random guessing. The dataset is split between 50% vulnerable and 50% non-vulnerable, so an accuracy of 50% of the model means that it is random guessing or choosing one of the two options for all data and getting 50% accuracy. The classifier IVDetect made by Li et al. (2021) was trained on the same dataset. Their model performed better than other state-of-the-art models on the same dataset. This shows that their IVDetect approach can learn the statistical differences between vulnerable and non-vulnerable code. The IVDetect model was trained on three different C and C++ datasets, the model performed the worst on the Big-Vul dataset. This indicates that the dataset is less suitable for deep learning than other datasets.

The results for the different CWEs of Table 7.3 and Table 7.4 show that the binary classifier is a little better with classifying some vulnerabilities types. The highest accuracy is achieved for the sequential LSTM classifier for CWE-404 (Improper Resource Shutdown or Release).



With an accuracy of 57.9% out of 19 cases. This accuracy is probably caused by chance and not an indicator of a significant difference of accuracy. This is confirmed by the accuracy of the tree LSTM classifier on this same vulnerability type. The tree classifier achieves an accuracy of 48,38% for this CWE, so not higher than the baseline of 50%. In the two tables it is also visible that the more times a vulnerability type had to be classified, the closer the accuracy reaches 50%. This indicates that the vulnerabilities where the classifiers had a higher or a lower accuracy than 50% will get closer to 50% after more tests.

The success of a machine learning model is not only caused by the architecture of a model. The data used to train the model is also important. The code in the Big-Vul dataset is a fragment of a larger program, as such it cannot be compiled. This could be the cause of the parsing problems with the Clang parser and the ANTLR parser. The `<ERROR>` tokens, used for mitigating parsing errors, allowed us to train the model on more data, but it might also hinder the data quality by removing important program information from the data. This might cause information to be missing in the dataset that would be needed for a model to learn to fix vulnerabilities. Another problem with the dataset is that a single vulnerability can span multiple fragments, so the fragment itself might not be vulnerable. A fix would be to combine all the fragments from a CVE and use that as input, downside is that this increases the input size. Looking at how successful other researchers were in using the dataset, the IVDetect model trained worse using the Big-Vul dataset compared with two other C/C++ datasets. The VRepair model only used the Big-Vul dataset for evaluation, so it is difficult to draw conclusions on the effect of the dataset on their model.

The models and the accompanying code that was created in this thesis will be useful for other researchers that also research automatic vulnerability repair. The code is publicly available on Github (<https://github.com/Jordi-klarenbeek/LSTM-Bug-Fixing>). The ANTLR parser for C might be an inspiration for researchers that also have to parse C or another programming language. The ANTLR language recognition framework is easy to learn, but still having example implementations are useful for understanding the framework.

# Chapter 9

## Conclusion

Problems with software can have a great impact on our life. Issues relating to the security of software can be devastating through ransomware or data leaks. Therefore it is important that the software we use is repaired of security vulnerabilities. Vulnerabilities are hard to define, a smart solution such as deep learning could be useful. Since deep learning allows the computer to learn the rules and definitions of a security vulnerability. Goal of this thesis was to explore deep learning models for repairing vulnerabilities in the C and C++ programming language, using the Big-Vul dataset (Fan et al., 2020).

The implications of our findings are that the vulnerabilities in the Big-Vul dataset are difficult to train a model on. The tree2seq model learns to a certain extent how to reconstruct the vulnerable functions into non-vulnerable functions, but it does not succeed in producing exact matches.

The results of the binary classifier show that it is difficult for a neural model to learn the statistical differences between vulnerable and non-vulnerable code in this dataset. This might have different reasons. The first reason could be that the fixes to the vulnerabilities are not as effective as expected by the programmers. The programmers that repair the vulnerabilities are often not the same as the security experts that found the vulnerabilities. Therefore the programmers might miss knowledge to prevent new vulnerabilities. The second reason could be that the vulnerabilities are not detectable since they are indistinguishable from non-vulnerable code. As described in the background section, vulnerabilities are often valid functionalities of a program that can be misused via a series of actions. These series of actions might be harmless by themselves, but result in a vulnerability if these actions are taken after each other.

Predicting the full program has been shown to be difficult, as the model needs to predict long sequences. A better approach would be to only predict the code that needs to change. The edit model tried to achieve this with the Github patches, the problem is that the Github patches contained a lot of duplicate code. Better preprocessing would be needed to remove the duplicate tokens and reduce the output sequence length.

A few possible changes to the current models could be to use an attention mechanism, a copy

mechanism for the <UNK> tokens and beam search for the patch interference. Currently greedy search is used, so selecting the token candidate which has the highest probability. With beam search multiple candidate sequences are created and the top candidate when the maximum sequence length is reached.

In the Big-Vul dataset there exist some non-vulnerable functions that have not been used in training the models. These functions might be used to pretrain the LSTM models. This pretraining might simply be the identity function or returning some tokens that are replaced with random tokens (to mimic completion and practicing code comprehension). In this manner the network will learn to produce functions before fixing vulnerabilities.

Another possibility would be to train the model on another dataset containing bugs and fixes. For example the SATE IV dataset (Okun et al., 2013), which contains over 120000 synthetic examples in the C and C++ programming language. Besides using an already existing dataset, we could also create our own dataset by searching on Github for git commits that mention "bug fix" or "repair", and use the pre and post code.

On the side of the neural network choice an improvement might be to use a transformer network or the network of the IVDetect model (Li et al., 2021). Transformer networks need a lot of data to learn properly, so only training on the Big-Vul dataset might not be sufficient to get a good functioning network. An option could be to use another pretrained transformer network such as codeBERT (Feng et al., 2020) and copy parameters from that model to initialize our model.

For the binary classifier an interesting avenue to pursue would be to classify which CWE vulnerability type a vulnerable code fragment is. Training a model only on vulnerable code fragments. The model would have difficulty learning less frequent vulnerability types, as it has less opportunity to learn about these types. Therefore it would be best to train this vulnerability type classifier on types that occur often in the dataset. For example CWE-119 (Buffer Overflow) or CWE-264 (Permissions, Privileges, and Access Controls).

# Bibliography

- M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.
- U. Alon, R. Sadaka, O. Levy, and E. Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.
- D. Alvarez-Melis and T. S. Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. 2016.
- A. Awasthi, S. Sarawagi, R. Goyal, S. Ghosh, and V. Piratla. Parallel iterative edit models for local sequence transduction. *arXiv preprint arXiv:1910.02893*, 2019.
- Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- B. Berabi, J. He, V. Raychev, and M. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.
- S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 2020.
- S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.
- Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.
- Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *arXiv preprint arXiv:2104.08308*, 2021.
- K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- J. Dawe. pytorch-tree-lstm. <https://github.com/unbounce/pytorch-tree-lstm>, 2019.
- J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

- J. Fan, Y. Li, S. Wang, and T. N. Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.
- Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, 2009.
- I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11): 139–144, 2020.
- R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaii conference on artificial intelligence*, volume 31, 2017.
- J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and P. Chin. Learning to repair software vulnerabilities with generative adversarial networks. *arXiv preprint arXiv:1805.07475*, 2018.
- J. Harer, C. Reale, and P. Chin. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*, 2019.
- S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.
- P. Kari. Who’s behind the kaseya ransomware attack – and why is it so dangerous? *The Guardian*, 2021. URL <https://www.theguardian.com/technology/2021/jul/06/kaseya-ransomware-attack-explained-russia-hackers>.
- S. Kim, J. Zhao, Y. Tian, and S. Chandra. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*, 2020.
- J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.
- C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.
- Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. *arXiv preprint arXiv:2106.10478*, 2021.
- B. Liu, Y. Liu, and K. Zhou. Image classification for dogs and cats. *TechReport, University of Alberta*, 2014.
- R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O’neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3):365–396, 2010.

- M. J. Michl. *Analyse sicherheitsrelevanter Designfehler in Software hinsichtlich einer Detektion mittels Künstlicher Intelligenz*. PhD thesis, Technische Hochschule, 2021.
- M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.
- L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- P. Murali, A. Sandur, and A. A. Patil. Correction of logical errors in c programs using genetic algorithm techniques. *International Journal of Recent Trends in Engineering*, 1(2):176, 2009.
- V. Okun, A. Delaitre, and P. E. Black. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication*, 500:297, 2013.
- K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.
- V. L. Shiv and C. Quirk. Novel positional encodings to enable tree-structured transformers. 2018.
- K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.
- F. Wotawa and M. Stumptner. A model-based approach to software debugging. *DBAI technical reports*, (TR-96-05), 1996.
- H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *arXiv preprint arXiv:2003.03296*, 2020.
- W. Zhao, L. Wang, K. Shen, R. Jia, and J. Liu. Improving grammatical error correction via pre-training a copy-augmented architecture with unlabeled data. *arXiv preprint arXiv:1903.00138*, 2019.
- Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.