Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Optimizing Vulnerability Detection with Natural Language Processing Techniques

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Levan Jandieri
geboren am:	26.08.1994
geboren in:	Tbilisi, Georgien
Gutachter/innen:	Prof. Dr. Lars Grunske
	Prof. Dr. Timo Kehrer

eingereicht am: .... verteidigt am: ....

Contents

1 Abstract

With technology being invariably intertwined in our lives and a high percentage of brands, government establishments and citizens relying upon it to store their data and process their tasks, the security of this huge net, especially of the algorithms and the data that it carries, is of utmost importance. Therefore, it is instrumental that vulnerabilities that allow certain proprietary software and websites to be exploited, are detected and dealt with in a timely fashion. Vulnerability detection has been a topic for decades and there have been various attempts to automate this process. Automating vulnerability detection is necessary mainly because of the human labor associated with designing the tools and the experiments and with manually defining features that help detect possible exploits; this is a tenuous and a taxing process and, worst of all, is neither efficient nor accurate in most cases. Furthermore, manual vulnerability detection tools, such as static analysis, code similarity analysis, can identify only the particular susceptibilities in the system that they were designed to find and do so in a time-inefficient manner.
As an alternative to classically manual tools, the idea was born of a natural language processing model trained on software code that could learn to identify vulnerable code without any pointers from humans. An example of this idea is training of a deep learning Long Short-term Memory (LSTM) model on source code with a fraction of vulnerable code. In such an experiment, a natural language processing model is trained on a text classification task on code snippets that are tagged either vulnerable or not vulnerable. This thesis attempts to improve the efficiency and the performance of deep learning models at discovering insecurities by applying different, more recent and potentially better natural language processing techniques in order to achieve a better F1 score.

2 Introduction

2.1 The problem and its setting

The International Organization for Standardization (ISO) defines a *software vulnerability* as “In the contexts of information technology and cybersecurity, a vulnerability is a behaviour or set of conditions present in a system, product, component, or service that violates an implicit or explicit security policy” in ISO/IEC 29147 [44]. As the global network of the Internet grows and more people engage in it, exploits in the system become more detrimental. Various examples illustrate this, such as Google+ network exposing personal data of about 500000 customers [4], the infamous EternalBlue exploit in Microsoft Windows operating system that was used by the crypto-ransomware WannaCry to spread itself in the network [8] and the *ShellShock* vulnerability that was present in almost all versions of Unix-based operating systems [5]. Such breaches not only harm the customers’ experience and jeopardize their privacy, but also cost significant financial damages to the companies that experience them. In a 2021 report, IBM computed ransomware breaches to cost approximately $4.4 million [6]. Global damages due to cybercrime in 2020 was estimated to be about $950 billion and the total spending on cybersecurity was approximately $150 billion [7]. This puts into perspective the significance of protecting software from vulnerabilities.
However, identifying and fixing these vulnerabilities is no easy task. It is virtually impossible to record or predict every kind of exploit; Common Vulnerabilities and Exposures (CVE) is the most comprehensive database with over 150 thousand entries in April 2022 [9]. But it is not complete and can never account for all possible vulnerabilities. Furthermore, not all companies require software developers to check their code for vulnerabilities that are present in such databases. On the user side, people choose not to update their systems and frequently use software that is out of date that leaves them susceptible to cyber attacks.
In order to detect vulnerabilities without relying on a third-party, code analysis tools have been developed and used. Static analysis is a source code evaluation technique, where the code is scanned without running it to find certain exploits according to a predefined set of rules [10]. In contrast, dynamic analysis is the examination and instrumentation of code execution in an attempt to identify flaws [11]. The two techniques complement each other and are commonly used in conjunction. Unfortunately these procedures are lacking - static analysis tools tend to tag non-vulnerable code as vulnerable and dynamic analysis tools often miss vulnerabilities that are present. The rise in popularity of Deep Learning and the overwhelming number of open source software have encouraged a more data science oriented approach to tackling this problem. Thus, there have been numerous efforts to build a comprehensive learning model that can detect crucial vulnerabilities in software. In recent years over 90 papers have been written that tried using a deep learning model based system for this purpose, some of them attaining over 90% accuracy. Nonetheless, the efficacy of this approach has the potential to improve by taking advantage of state-of-the-art architectures in deep learning, such as Transformers.

2.2 Goal and structure of this thesis

The goal of this thesis is to build upon and enhance the performance of *Vulnerability Detection with Deep Learning on a Natural Codebase* (VUDENC) [14] by applying newer or potentially superior Natural Language Processing techniques, namely Character Level Models and Transformers. For comparison purposes, these techniques will be used on the same database with the same vulnerability types as in the work by Wartschinski et. al.

The remainder of this thesis is structured as follows. First, the related work is discussed in Section 2. Section 3 introduces preliminaries and the background on the topics relevant to this thesis, such as natural language processing, deep learning, model training and testing. In Section 4, the approach of deploying a character-level LSTM and Transformer models is presented, followed by its implementation in Section 5. Section 6 is the evaluation of the experiments and Section 7 is the discussion of the results. The thesis concludes in Section 8.

3 Related Work

This section discusses the works related to the topic of this thesis and the state-of-the-art solutions to the problem presented by it. Specifically, it focuses on current vulnerability detection techniques and their efficiency at this task.
Up until recently, vulnerability detection in software was almost exclusively done by static analysis tools. Some of these were commercial, while others were open source and academic projects. Some more in-depth tools incorporated data flow and control flow analysis of code [11]. Others [12] were designed to choose one from a list of predefined algorithms to handle a particular type of vulnerability.
More recent attempts to create vulnerability detection frameworks built upon the advancements in natural language processing, more specifically, in deep learning. Neural network structures that mimic the behavior of neurons in the human brain were used to learn to identify vulnerabilities in source code. Long short term memory in [14] and bidirectional long short term memory [13] were used to achieve this goal, with impressive results.
VulPecker [12] is a code-similarity based vulnerability detection tool. It features a training and a detection phase. The training has three inputs: National Vulnerability Database (NVD), Vulnerability Code Instance Database (VCID) and Vulnerability Patch Database (VPD). For a vulnerability entry in the training dataset, basic features of the vulnerability are derived from NVD, a patched version is obtained from VPD and the code reuse examples from VCID. VulPecker uses this information to determine the most efficient code-similarity algorithm for the particular vulnerability type. The end result of the training phase is a mapping of Common Vulnerability and Exposure IDs (CVE-ID) to appropriate code-similarity algorithms. Incoming data for the detection phase to function are CVE-IDs, the the code of the software to be analysed and the CVE-to-algorithm mapping trained in the first phase. Using the CVE-ID and the mapping, a code-similarity algorithm is chosen to detect vulnerabilities on the source code. The F-measure of the experiments conducted using VulPecker was reported to be just over 90%.
VulDeePecker [13] was one of the first proposals of using deep learning to tackle the problem of vulnerability detection. The vulnerability detection system was implemented in 2018 and focused on three research questions:

- Can it detect different kinds of vulnerabilities?
- Can humans supervise the system and therefore improve its efficiency? As opposed to a fully automated system.
- How does this particular system perform relative to other state-of-the-art systems?

The system uses bidirectional long short-term memory (BLSTM) models to be able to predict vulnerabilities. To be able to use the code as input for a BLSTM model, the authors had to somehow convert it into vectors. This was achieved by reshaping the program into code gadgets, defined as several lines of code that are connected to each other. After the code gadgets are assembled they are converted into *symbolic representations* in order to preserve a certain amount of semantic information of the original program. Finally, the symbolic representations are transformed into vectors that are fed to the BLSTM model.

The operation of VulDeePecker is divided into two phases: training and detection. In the training phase, a bulk of code examples is fed into the BLSTM. Certain percentage of these contain vulnerabilities. Such examples are labeled as vulnerable so that the model can train and learn patterns from them. The detection phase utilizes the trained model to detect vulnerabilities in programs that were not used in the training phase. In both phases, code gadgets are produced to modify the programs into appropriate vectors that can be read by the BLSTM.
With optimized hyper-parameters, the system was able to achieve F1 scores ranging from 75 to 95 on various datasets. However, a severe limitation of the study was that only two types of vulnerabilities were considered: Buffer vulnerabilities and Resource Management. Furthermore, only code considered was Library/API Function Calls. However, the research questions posed were answered as follows:

- A BLSTM model can be trained to detect multiple vulnerabilities. This is illustrated by using the model on a dataset with two different types of vulnerabilities and achieving approximately 80% F1 score with optimized hyper-parameters.
- Humans can supervise the model by preprocessing the data and using heuristics to help the model label input programs.
- VulDeePecker did outperform other systems that were relevant at that time. For instance, it almost doubled the F1 score of Checkmarx [32], a vulnerability detection static analysis toolkit.

VUDENC [14] is a tool that utilizes deep learning to find vulnerabilities in python code. It employs LSTM models that are trained on Python code aggregated from various open source projects on Github. In order to identify and train on vulnerabilities present in the source code, commits are gathered that contain specific vulnerability types as tags in the commit message. Since LSTM models require vectors as input, python code has to be parsed into vectors. The first step in VUDENC detection procedure is to train a word-to-vector embedder. This is technically implemented by training Gensim [33] word2vec on a large amount of python code. Result is a model that can convert python tokens into vectors. The input code is appropriately tokenized for the gensim model to be able to embed it. Seven different vulnerability types were examined in total.
The examined vulnerabilities are fairly frequent and the commits that fix them serve as favorable training data for the models. Note that each vulnerability type has a distinct

dataset and a separate model was trained on each of them. Several hyper-parameters, such as the number of epochs, number of neurons, dropout rate, were optimized by brute force. VUDENC achieves F1 scores and accuracy of between 80-87% and 92-96% respectively, depending on the vulnerability type. However, there are certain limitations of the study, such as: reliance on commit tags to identify vulnerabilities, focus on specific vulnerability types, negligence of vulnerabilities that were not detected by developers and thus never fixed.

4 Background

This section introduces preliminaries and terms that are referred to throughout this thesis. Namely, it defines metrics that are used to measure the efficacy of the experiments and explains important concepts such as Natural Language Processing, Neural Networks, Deep Learning, Long Short-Term Memory and Transformers.

4.1 Preliminaries

Confusion Matrix [1], also known as Error Matrix, is a table that represents the performance of algorithms, in this case the adaptness of vulnerability detection tools. It consists of four values that help understand how accurate the tool is and what percentage of vulnerabilities were discovered. The values are as follows:
True Positives (TP): Number of inputs correctly classified as vulnerable.
False Positives (FP): Number of inputs incorrectly classified as vulnerable.
True Negatives (TN): Number of inputs correctly classified as not vulnerable.
False Negatives (FN): Number of inputs incorrectly classified as not vulnerable.
Accuracy: Ratio of the total number of correctly classified inputs divided by number of all inputs.

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$$

Precision: Ratio of true positives divided by all inputs classified as vulnerable. This value demonstrates what percentage of tagged vulnerabilities are actually vulnerable. The closer it is to 1 the better for the system.

$$Precision = \frac{TP}{TP+FP}$$

Recall: Ratio of true positives divided by the amount of all vulnerable inputs. It shows how many vulnerable inputs were missed by the detection tool. This value should also be optimized to reach 1.

$$Recall = \frac{TP}{TP+FN}$$

Both Precision and Recall present significant insights into the performance and are frequently used to evaluate machine learning classifiers. However, an even more useful metric is the F1 score.

F1 Score: Harmonic mean of the precision and the recall scores, therefore it takes into account falsely classified vulnerabilities as well as the vulnerabilities that were skipped by the system and will only become near 1.0 if both precision and recall are high. F1 score of 1.0 means that there are zero undetected vulnerabilities in the code and there were no false positives.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

4.2 Natural Language Processing

Natural Language Processing (NLP) is an area of research that stems from linguistics. It tries to train computers to understand and imitate human behavior in respect to natural language for the purpose of manipulating them to perform several tasks, such as text processing, summarization, translation, information retrieval [2]. Over the past two decades NLP has been increasing in popularity and used in more consumer products, such as virtual assistants and translators; this progress has been fueled by several advancements: more powerful computers, copious amounts of language data, more complex machine learning techniques and improvements in our understanding of human language [3]. NLP has always been a branch of Artificial Intelligence; the idea was first conceived in 1950 by Alan Turing in his publication *Computing Machinery and Intelligence* [19] as the Turing Test. In the work Turing proposed The Imitation Game, where machines are set to perform NLP tasks, such as natural language generation and interpretation in order to be indiscernible from humans. Although these tasks fell under today's definition of symbolic NLP, the idea was then not yet classified as part of Natural Language Processing.
Earlier NLP models were symbolic, rulebook systems, based on Noam Chomsky's generative grammar introduced in 1957, where the rules had to be defined by humans. Famous examples include The Georgetown Experiment (1954) [20] and ELIZA by Joseph Weizenbaum [21]. In the 1980s with the rising popularity of machine learning, handwritten rulebook-based models quickly got replaced by machine learning models. The field was further improved in 1990 when IBM researchers acquired a large amount of English-French sentence translations.
Today most NLP models are neural network based. Neural networks are powerful since they can better imitate human behavior. They are used for various goals: messaging with voice, filtering spam, spell checking, autocompleting queries, keyword searching via search engines, voice assistants such as Siri, Alexa, Cortana. Aside from the metrics defined in section 3.1, there are several common measures that serve to demonstrate the efficiency of such models: Mean Average Precision (MAP) score, Mean Absolute Percentage Error (MAPE) score, Bilingual Evaluation Understudy (BLEU) score, Root Mean Squared Error (RMSE) score and General Language Understanding Evaluation (GLUE) benchmark. GLUE is one of the most commonly used and important metrics for measuring NLP; there is a regularly updated leaderboard of the NLP models with the best GLUE scores [22]. A significant percentage of the top performing models are transformer based BERT variations. These models are certainly powerful, but there are certain drawbacks. One major limitation of NLP is that most resources are available for only a small subset of languages, such as English, German, French and Chinese. Other shortcomings are models' inability to identify irony and sarcasm, the ambiguity regarding homonyms and phrases, and the confusion involving colloquialisms and slang.

4.3 Neural Networks

Artificial Neural Networks, or cited simply as neural networks, are a series of algorithms that imitate the behavior of neurons in the human brain. The main goal of this process is to identify and recognize underlying relationships in a complex set of data and train weights that correspond to them. Such a network is ideally first trained on a large dataset. Neural network contains certain amount of neuron cells, also called nodes, that are distributed over a number of layers. Every neuron has individual weights that have been fine tuned during a training process. Upon receiving inputs, neurons combine them with weights that determine which parts of the data are significant, and aggregate the result into a net input function that consequently goes through an activation function to generate the final output of the neuron, which is a single number [30]. This process is referred to as neuron's activation.
Since most neural networks have more than one layer, the inputs are passed through a series of neurons that are interconnected, starting with the input layer, followed by a number of hidden layers and the result at the output layer. Through these connections, neurons are able to observe when other neurons are activated. As a result, individual neurons can choose to activate or not to activate according to the behavior exhibited by the neurons before, sequentially determining the activation of the neurons in the next layers, until operation concludes at the output layer. The term dense layer refers to a layer of neurons where each neuron is connected to every neuron to the previous layer. Deep learning means simply a large amount of neural layers present in the model architecture.

4.4 Long short term memory network (LSTM)

Long short term memory networks are special kinds of neural networks that were designed as a solution to the vanishing gradient problem in NLP. LSTMs were first introduced in 1997 by Hochreiter and Schmidthuber in [23]. In the vanishing gradient problem, the information saved in the neural network continuously goes through a series of layers that multiply it by small numbers, resulting in significant information loss. LSTM handles this by incorporating a more long-term memory layer called the cell state that accumulates information over time. Furthermore, the model has so called 'neural gates' that control the information flow: the information going through the gates are multiplied by a number ranging from 0, meaning deletion of information, to one, meaning conserving all of the information. The gates are neural layers and, as such, are trained and have their own weights.
LSTMs have become increasingly popular in the last decade. They have been used in countless tasks for huge projects, such as Apple's Siri [24], Google's Image Caption Generation [25], Samsung's Audio Visual Speech Recognition [26], Amazon's Alexa [28] and IBM's usage of LSTM to analyze emotions [29].

4.5 Transformers

Transformer is the state-of-the-art natural language processing technique first introduced in 2019 by Gomez et al. [15]. The novelty of the transformer architecture resides in its residual attention mechanism and, unlike the long short-term memory, the transformer has the ability to process input sequence without recurrence. As such, transformers require significantly less processing time and are highly parallelizable.
Transformers are applied on a variety of NLP tasks, including, but not limited to:

- **Text analysis tasks**: classification, translation, summarization, information extraction, question answering, text generation.
- **Audio analysis tasks**: classification and speech recognition.
- **Image analysis tasks**: classification, object detection.

Attention is a NLP mechanism that helps the model discover associations of words within the input sequence. As the model processes a particular word, it simultaneously looks at other positions in the sequence for clues that can help it encode the current word better.
Technical implementation maps a query to a set of key-value pairs to an output. The query, key and value are weighted matrices that have been trained. The input sequence is projected into query, key and value vectors using trained weight matrices W_{query}, W_{key} and W_{value}.

4.6 Transformer Architecture

The transformer implemented in *Attention is all you need* featured two key parts: Encoder and Decoder. Note that this work will refer to this particular implementation of the transformer as the original transformer.

Encoder. The purpose of the encoder is to transform input sequence into a contextualized sequence. In other terms, the encoder preprocesses and prepares the sequence for further processing and classification by the decoder. The encoder in the original transformer consisted of six identical layers. Each of these layers contained two sub-layers: a multi-head self-attention layer and a feed forward neural network.

Decoder. The purpose of the decoder is to calculate the probability distribution of the input sequence given the contextualized encoded sequence. Or, in other terms, decoders are the parts of the transformer architecture that are responsible for classifying the sequence. The decoder in the original transformer contained six identical layers. Each of these layers contained three sub-layers: a self-attention layer, an encoder-decoder attention layer and a feed-forward neural network.

Self attention layer. Sub-layer that focuses on the current word in relation to all other words in the sequence and, using attention, encodes the current word.

Encoder-decoder attention layer. Sub-layer that applies the attention mechanism on the output of the encoder stack.
Feed-forward neural network. Sub-layer that computes an approximate function for the fixed size input sequence.

Note that the number of identical layers in the encoder and the decoder - six - does not bear significance. It is not an optimized number, but was chosen instinctually.

4.7 Pre-training of Deep Bidirectional Transformers for Language Understanding (Google BERT)

The key design decision of the BERT transformer is to pre-train representations bidirectionally from unlabeled text by jointly conditioning on both left and right context in all layers. The key design decision of the BERT transformer is to pre-train representations in two directions from untagged data by considering context on each side of layers simultaneously. The model is pre-trained on two unsupervised tasks instead of pre-training using classic left-to-right or right-to-left models. The first task is Masked Language Modeling (MLM). Its purpose is to train the model bidirectionally. MLM involves masking part of tokens and using the model to predict the masked tokens. During this task up to 15% of the tokens are masked - 80% of the masked tokens are replaced with placeholder token '[MASK]', 10% with a random word from the vocabulary and 10% are left untouched. The second task in the pre-training is Next Sentence Prediction (NSP). It consists of a binarized next sentence prediction pre-training; each input contains a pair of sentences A and B, where A is the preceding sentence and B is the correct next sentence half of the time, labeled `isNext`, or is an incorrect next sentence, labeled `NotNext`. The model learns to predict the correctness of succeeding sentence B. With one additional layer, BERT transformer was able to outperform state-of-the-art models for a wide range of NLP tasks. It achieved a GLUE score of 80.5%, MultiNLI accuracy of 86.7%, SQuAD v.1.1 score of 93.2 and SQuAD v2.0 score of 83.1 [16].

4.8 A distilled version of BERT (DistilBERT Transformer)

DistilBERT is a successful attempt at using Knowledge Distillation, a method to decrease the size of the model, to condense the BERT architecture to a smaller model that is easier to operate and handles computational training constraints. The result is a pre-trained language representational model for a wide range of tasks. The general architecture of the model is the same as that of the BERT. However, some components, such as token-type embeddings and the pooler are deleted. Moreover, the main target of the distillation is the amount of linear layers, since it has the most impact on the efficiency. The evaluation showed a 40% decrease in model size and an increase of 60% in operational speed, whilst preserving 97% of the performance [17].

4.9 A Robustly Optimized BERT Pretraining Approach (RoBERTa)

RoBERTa is a reproduction study of the BERT model that rigorously evaluates the effect of hyperparameter tuning across various pre-training data sizes. The idea was born when the authors of RoBERTa indicated that the original BERT was not sufficiently trained [18]. The model is optimized through a series of techniques: dynamic masking, entire sentences without loss in next sentence prediction, higher batch size and higher byte-level encoding. Two other aspects are the amount of data used for pre-training and the amount of training epochs. Consequently, it achieves 94.6, 89.4, 90.2 and 96.4 scores on SQuAD 1.1, SQuAD 2.0, MNLI-m and SST-2 respectively, outperforming classic BERT across all the metrics. However, it should be noted that the mentioned achievements were possible only by training the model on 160GB of data, using a batch size of 8K, as opposed to 13GB and 256 for BERT.

5 Approach

In this section I will present my approach of implementing character-level LSTM models and transformer-based models, training them and testing their efficiency on the Python dataset. I start by a short examination of the dataset, granularity, embeddings and the architectures of the models.

5.1 The data

The dataset used in this work was gathered on Github for VUDENC [14]. Various Github repositories were scraped for commits that fixed one of the target vulnerabilities. A critical advantage of using Github as a data source is that the projects covered are actively used in practice, as opposed to artificially generated datasets that are commonly used for code analysis. Furthermore, there are countless relevant open source code repositories that inflate the size of the training corpus and create a dataset that comprises of more vulnerability examples. This helps train a more comprehensive model.
In addition to the database for training LSTM and Transformer models, a separate codebase was mined from Github for the purpose of training Word2Vec embedding, used for Char2Vec embedding in this work.
The VUDENC dataset consists of seven distinct subsets each representing a particular type of vulnerability. The names of the subsets and their abbreviations are:

- SQL Injection (SQL)
- Cross Site Scripting (XSS)
- Command Injection (CI)
- Cross-site request forgery (XSRF)
- Remote code execution (RCE)
- Path disclosure (PD)
- Open redirect (OR)

5.2 Granularity

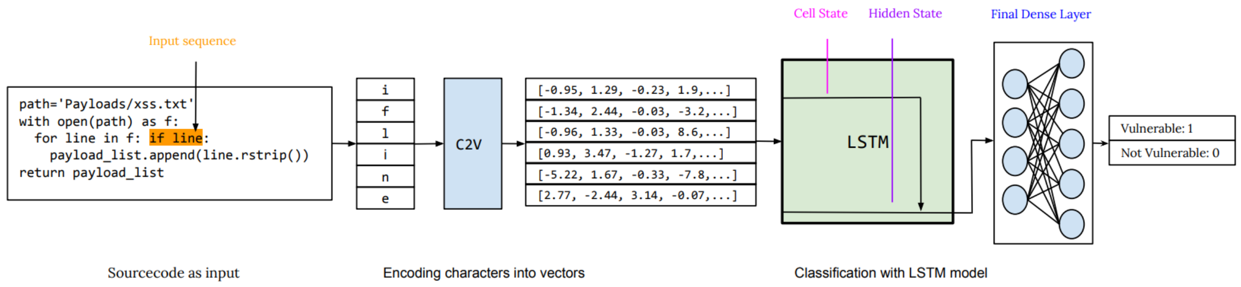
Choosing the right granularity level is important because it determines both how well a neural network model would perform and how useful it would be in terms of locating the vulnerability precisely and fixing the issue. Intuitively, binary source file level granularity might be enough to locate particular binary files that are vulnerable, but it does not precisely locate the areas of code that need to be inspected [40]. On the other hand, exceedingly high granularity, where a model would be trained to detect vulnerable binary code, could theoretically be precise, but its usefulness would be questionable, since developers would not be able to understand what causes the vulnerability in the binary code or how to patch it. Thus, instruction-level granularity was chosen as an optimal middle-ground between the other options.

5.3 Embeddings

For the chosen models, an embedding is necessary to convert natural code into vectors so that they can be processed by neural networks. Although word level models make more sense in this context, character-level models have been shown to be quite effective and outperform, but can be influenced by a series of conditions, such as database size, alphabet and text curation [39]. Thus, the LSTM architecture employs a character-to-vector embedder as an attempt to capture features that could be missed out when staying at token level. As for the embedding for transformer models, a simple tokenizer was used. One of the central ideas of the attention mechanism in the Transformer model is to draw connections between the elements of the input sequence. Particularly, unlike LSTM models, Transformers can efficiently detect long-range dependencies. Such dependencies are most commonly found between Python tokens and not between characters. For this reason, it makes more sense to choose a token-level embedding.

5.4 Architectures

The first approach realized in this thesis is an LSTM model with Character-to-Vector encoding. The code is split sequences of certain window size and then into individual characters and each character is embedded into a vector of specified size. The vectors are loaded into the LSTM structure and trained on them in order to identify vulnerable code through learning the features of the input. The second approach is the transformer

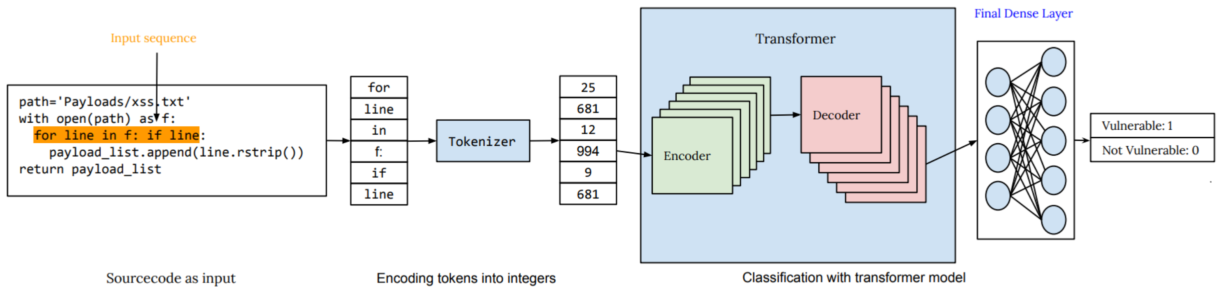


```
path='Payloads/xss.txt'
with open(path) as f:
  for line in f: if line:
    payload_list.append(line.rstrip())
return payload_list
```

```
[-0.95, 1.29, -0.23, 1.9,...]
[-1.34, 2.44, -0.03, -3.2,...]
[-0.96, 1.33, -0.03, 8.6,...]
[0.93, 3.47, -1.27, 1.7,...]
[-5.22, 1.67, -0.33, -7.8,...]
[2.77, -2.44, 3.14, -0.07,...]
```

Figure 1: Character-level LSTM architecture

model that is realized similarly. But instead of embedding characters, now a tokenizer is used to split text into Python 'words' and encode them into their corresponding integer values. These code representations are then embedded into a vector of specified length by the first encoder and then further processed by the transformer block. The final dense layer outputs a single value that represents the vulnerability of the input sequence.



```
path='Payloads/xss.txt'
with open(path) as f:
  for line in f: if line:
    payload_list.append(line.rstrip())
return payload_list
```

Figure 2: Transformer Architecture

6 Implementation

6.1 Acquiring the data

The data used in this work was collected in the project VUDENC [14]. It was scraped on GitHub using various commit keywords, such as ‘buffer overflow’, ‘remote code execution’, ‘cross site request forgery’, to find commits that fixed vulnerabilities. The chosen repositories were marked and some of them were filtered out since they did not fix vulnerabilities, but instead introduced new security measures against them. Once the filtering was done, the ‘diff’ files, referring to the text files that represent the changes made in the code, were downloaded. Only the diffs that affected Python files were taken into account. Furthermore, the downloaded diffs were searched for duplicates and they were removed. The data was filtered more by removing files that contained embedded HTML or used Sage system [43], since these might have introduced vulnerabilities that are not Python-based and not optimal for the model to learn. Finally, all comments are removed from the source code, since they do not affect the potential vulnerability of the code and provide little predictive insight.

The Python codebase that was used to train char2vec embedding was acquired and pre-processed in a similar fashion. Except, in this case another decision had to be made: to keep or remove string values. After comprehensive testing, it became evident that char2vec models that did not disregard string values increased the performance of LSTM models.

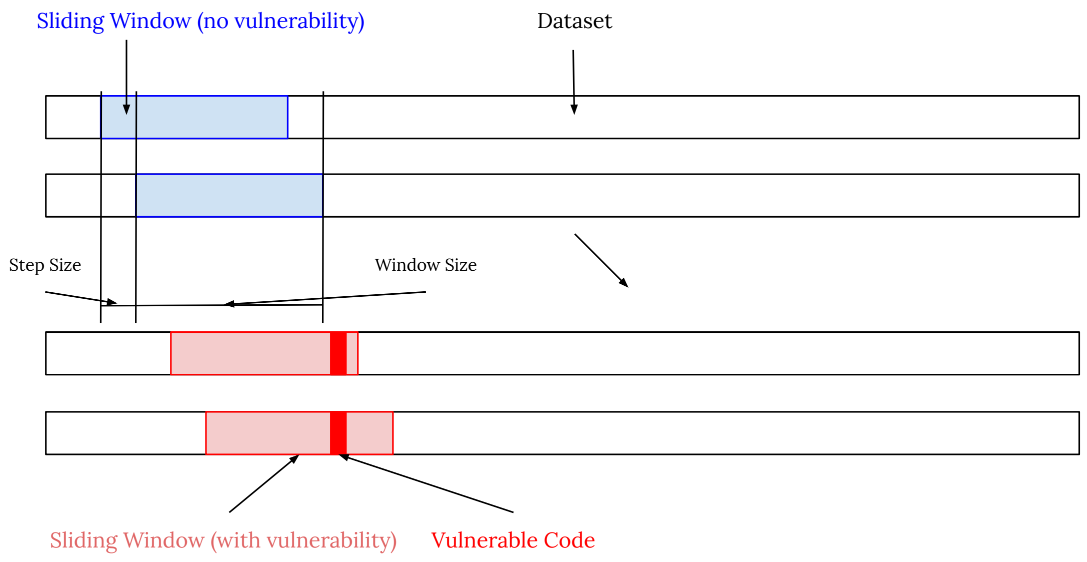
The data for char2vec and neural network training were downloaded and can be found at [41] and [42] respectively.

The Python codebase that was used to train char2vec embedding was acquired and pre-processed in a similar fashion. Except, in this case another decision had to be made: to keep or remove string values. After comprehensive testing, it became evident that char2vec models that did not disregard string values increased the performance of LSTM models.

6.2 Preparing the data

In order to generate input sequences from the data, a sliding window of fixed size was used to move over it. The window was shifted by a specified step size. However, the edges of the window were snapped to the start and the end of tokens. As such, the database was divided into overlapping strings of equal size. Each sequence was then labeled as vulnerable if it contained a subsequence that was patched on Github. Both the window size and the step size can be altered and have a significant impact on the models that are subsequently trained on data defined by them.

Figure 3: Generating and labeling training sequences with a sliding window


Sliding Window (no vulnerability)
Dataset
Step Size
Window Size
Sliding Window (with vulnerability)
Vulnerable Code

6.3 Character-level LSTM Architecture

The character-level LSTM was implemented as a proof-of-concept based on the techniques described in Section 4. The model requires a corpus to train for character-to-vector (char2vec) embedder in order for it to be able to convert source code characters into vectors. The vectors are the only possible input to an LSTM model. The dataset used for training the LSTM model was taken from VUDENC. It is divided into seven vulnerabilities. An LSTM model was trained for each of the seven vulnerabilities, using the classic NLP partitioning of 70-15-15% training, validation and testing respectively. Keras Framework [34] was used to implement the LSTM models and train them on the dataset. It allows several hyperparameters to be set and outputs training sequences as well as the model performance in the console. The model is initialized as an instance of Keras Sequential, a class that groups linear sequential models. Certain conditions and hyperparameters have to be defined during the instantiation, as well as appropriate setup for GPU to be used. A dense activation layer is added to the LSTM and the structure is then compiled. The model is trained, validated and tested on a given vulnerability database and accuracy, precision, recall, F1 scores are displayed per data partition. The final dense activation layer outputs a single value, either 0 or 1, that classifies the input as either vulnerable or not vulnerable.

6.4 Transformer Architecture

The original Transformer is also implemented with Keras Framework. However, instead of a char2vec embedding, a text tokenizer was used from Keras.preprocessing module in conjunction with a dense embedding layer. The Transformer is realized through a Keras Sequential model that contains the following layers:

- Input Layer
- Positional and Token Embedding layer
- Transformer Block
- Global Average Pooling Layer
- Dropout Layer 1
- Dense ‘Relu’ Activation Layer
- Dropout Layer 2
- Dense Sigmoid Activation Layer

Transformer Block consists of self-attention layer, dropout layer, normalization layer, feed-forward network layer, another dropout layer and another normalization layer. Similar to character-level LSTM, transformers also need to be trained. Both the tokenizer and the model itself is trained on the vulnerability datasets. Same 70-15-15 split is used and a model is generated for each type of vulnerability. The main idea is the same: output a number: 0 or 1 to mark the input as vulnerable or not.

6.5 Transformers on multiple vulnerability subsets

After testing models on vulnerability subsets individually, it was time to train models on all subsets. The **first approach** to this task was to split each subset into 70-15-15 parts and then combine all training subsets into a large training corpus and train a transformer model on it. The corpus was shuffled to avoid having all sequences of certain vulnerability consecutively. The validation and test datasets of all vulnerability types were similarly combined and the model was tested on them.

The **second approach** was slightly different. Instead of partitioning the data, first all of the subsets were combined into one database. Then this database was split into 70-15-15 and transformer models were trained, validated and tested. Notice that this approach, unlike the first one, does not guarantee that each subset was equally represented or properly split on any of the training, validation or test chunks.

6.6 Architectures of BERT-based Transformers

BERT Transformer is implemented using the Flair framework [35]. The framework uses the HuggingFace version of the BERT transformer available at [36]. Normally, before initializing the model with this framework, a label dictionary is assembled from the data. In this case, the labels are 1 and 0 for vulnerable and not vulnerable respectively. The initial embedding used for this experiment is a pre-trained DistilBERT embedding. But the embedding is then fine-tuned on the datasets. After this, a text classifier model is initialized and trained on the vulnerability data.
The model consists of 11 Bert Layers each containing the following components:

- BertAttention Layer with biased query, key and value linear layers, followed by a dropout layer
- BertSelfOutput Layer with dense linear layer, normalization layer and dropout layer
- BertIntermediate Layer with dense linear layer and a Gaussian Error Linear Units (GELU) activation function
- BertOutput layer with dense and normalization layers with a dropout layer at the end

DistilBERT and RoBERTa models were also implemented via Flair using HuggingFace pre-trained models at [37][38]. The procedures of training the models are identical to the one of the BERT Transformer.

DistilBERT model contains five Transformer blocks, each made up of two main components - Multi-head Attention and Feed-forward Network.
Multi-head Attention has:

- dropout layer
- query linear layer
- key linear layer
- value linear layer
- output linear layer

Feed-forward network (FFN):

- Dropout layer
- Linear layer 1
- Linear layer 2
- GELU Activation Layer

There is a normalization layer between Attention and FFN layers and another normalization layer after the FFN.
The RoBERTa model incorporates 11 Roberta Layers and a pooler layer with hyperbolic tangent activation at the end. Roberta Layer architecture is defined as follows:

- Roberta Attention with query, key, value linear layers followed by dropout
- Roberta Self Output with dense and normalization layer followed by dropout
- Roberta Intermediate layer with one dense layer and GELU activation
- Roberta Output layer with dense and normalization layer, followed by output.

Although the RoBERTa model is identical to BERT in terms of structure, it has diverse values for parameters and is pre-trained on separate data, and as such, performs differently.

7 Experimental Evaluation

In this section the performance of this approach is evaluated by performing experimentation on the same dataset used in VUDENC and this approach is compared to the approach in VUDENC. I ask the following research questions:
RQ1 How effective are character-level LSTM and Transformer models at detecting vulnerabilities?
RQ2 Is it possible to optimize the VUDENC tool and improve its F1 efficiency by applying different and more recent deep learning techniques?
RQ3 Is it possible to train a model that can detect multiple vulnerability types? How efficient is such a model?

To answer these questions, I conducted an experiment set on a standalone computer with the following specifications.

- CPU: Intel i3-12100, 4 Cores, 3.3-4.3 GHz
- Operating Memory: 32 GB
- GPU: NVIDIA RTX 3080 12GB
- Operating System: Windows 10
- Python Version: 3.9

The data used was the same as in VUDENC. Table containing specific information about the data follows (Table 1). As per standard, data was shuffled each time a model was trained as to avoid the models learning the order of the data and to prevent bias. The downside of this approach is that there is certain amount of variance when training exactly the same type of model on exactly the same dataset.
The results were evaluated using the F1 metric.

Vulnerability	Repositoris	Files	LOC	Chars
sql injection	336	657	83558	3960074
xss	39	81	14916	736567
command injection	85	197	36031	1740339
xsrf	88	296	56198	2682206
remote code execution	50	131	30591	1455087
path disclosure	133	232	42303	2014413
open redirect	81	182	26521	1295748

Table 1: Dataset specifications

7.1 Character-level LSTM performance

Parameters used for this model were:

- 200 Neurons
- 35 Vector Size
- 0.1 Dropout
- 200 Epochs
- adam optimizer
- 128 Batchsize

There was a wide variance between vulnerability subsets for this model, ranging from 63.1% for SQL injection to 91.1% for remote code execution. The average was calculated to be 76.9% It seems that there were no consistent dependencies between characters of the input sequence across vulnerability subsets and the performance of most individual subsets actually diminished compared to the original Word2Vec approach in VUDENC, except on the RCE subset.

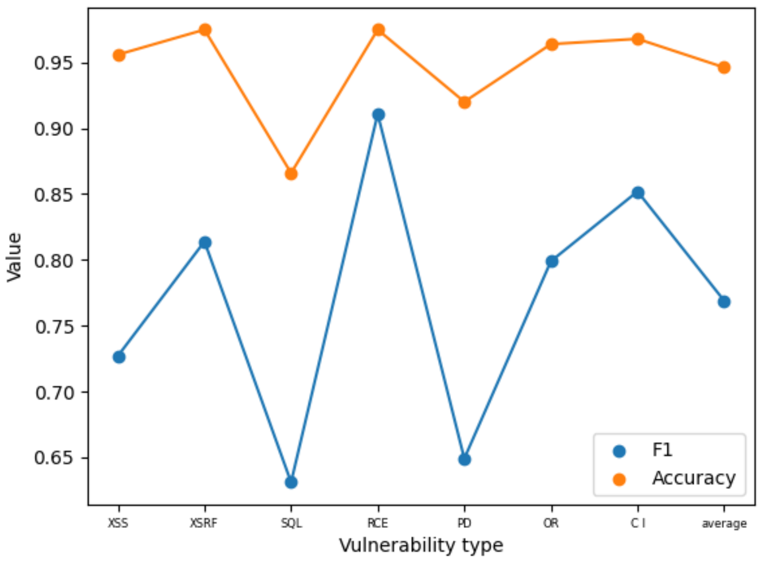


Figure 4: Character-level LSTM performance

7.2 Baseline Transformer Model

A baseline model was defined and used in order to individually alter its hyperparameters and illustrate the difference they make in the model's performance. However, the hyperparameters initially chosen for the baseline model are not arbitrary - they have been estimated to be optimal before actually testing them individually. The baseline model:

Parameter	Value
Epochs	100
Batch size	1024
Vocabulary size	30000
Window length	200
Step size	5
Embedding Dim.	32
Num. Additional Heads	5
Dropout	0.1
Optimizer	adam

Table 2: Set of parameters for baseline model

7.3 Optimal Hyperparameters for Transformer Models

The following experiments keep all hyperparameters of the base (keras) transformer model intact, only altering one of them at a time to separate their influence on the model.

7.3.1 Parameters for Tokenizer

Filter: At first glance, filtering certain characters from the source code makes no sense, since they might be parts of tokens or essential for the code to function. However, not all characters contain semantic information; some of them, such as ' ' (space), serve mostly for separating tokens, but sometimes are used on empty lines and do not add any content to the program. This particular problem was already initially handled in [14]: leading and ending space characters were stripped from the codebase.
Despite this, some characters remained that could be beneficial for the model to ignore. After setting a filter on the tokenizer, it completely ignored two prevalent characters: '\n' (newline) and '\t' (tabulator). This meant that, even though the model was presented with sequences containing these characters, the tokenizer chose to completely ignore them when dividing sequences into known Python tokens. Even though on paper this does not seem to be an important change, in practice it made a significant difference in the performance of the model: the baseline model went from 93.4% F1 score to 95.5%.

Vocabulary Size The vocabulary size determines the maximum amount of tokens that can be identified and stored by the keras tokenizer. Naturally, more tokens stored means better performance of the model, since it needs to identify all the tokens that are present in the input sequence. Accordingly, the F1 score of the model increases from under 87% to about 96% with vocabulary sizes of 5000 and 20000 respectively. Very small benefit is derived from increasing the vocabulary size past 80000. However, this number might be specific to the database and will underperform on databases that have a number of unique tokens that is significantly higher than 20000. Even so, 20000 is the optimal vocabulary size for the purposes of this work.

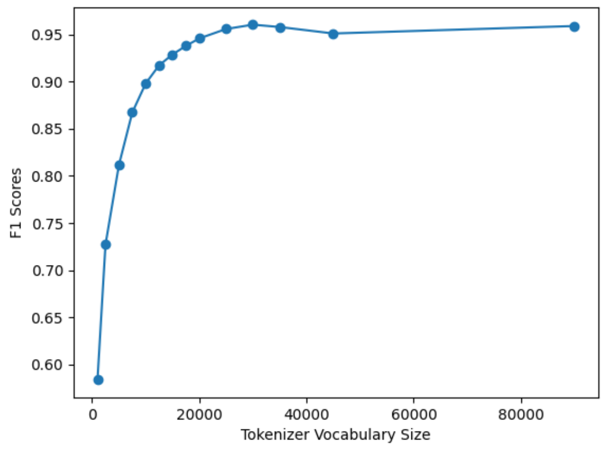


Figure 5: Optimal Vocabulary Size

7.3.2 Window Length

Window length is the amount of characters in an input sequence before it is converted to a vector and input into the transformer. Several sequence window lengths were tested to find an optimum. Since Transformers are capable of understanding and advantageous for processing contextual information, the smallest sequence length used was 10 characters so that at least 1-2 Python tokens were present in the sequence. This hyperparameter had a significant effect on the performance of the model, starting at the F1 score of approximately 52% at sequence length of 10, but rising up to around 96% at 500 length. The results plateaued with window length larger than 500; doubling it to 1000 yielded only a 0.3% increase in F1. There are certain considerations when choosing the sequence length, such as **overlap**, **context** and **precision**. With larger window length, there is a significant amount of overlap; for instance, given a vulnerable token of size 10, step size set to 10 and window length set to 1000, up to 100 consecutive sequences would have to be marked vulnerable because of this single token. On the other hand, having a window length that is sizable helps identify vulnerabilities that are caused by several tokens that

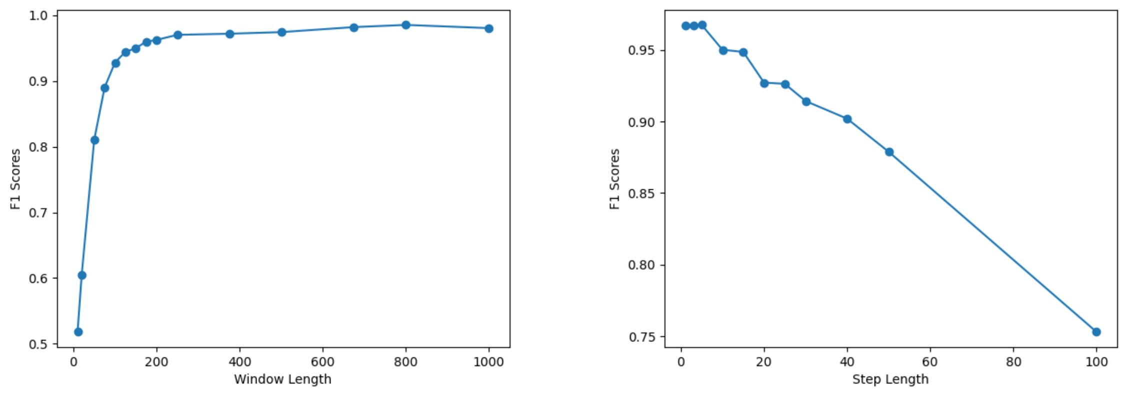


Figure 6: Window Length and Step Length

are distributed at different places in the code. Finally, even if a large sequence is correctly identified as vulnerable, it is harder to precisely locate which of the many tokens are causing the vulnerability.
Considering all this, 250 was chosen as the optimal value as it offers enough context and efficiency, but does not cause significant overlap or precision loss.

7.3.3 Step Length

In this context, step length determines how wide apart the sequences are. Smaller step length means more samples and more data for the model to train. However, a step size that is too small might promote overfitting.
The visual evaluation of the figure 6 shows that the lower the step length, the better the baseline model performs. The F1 scores range from 75% with step size 100 to 98.5% with step size 1. Note that having a step size of one generates duplicate sequences as input for the model, since sequences never slice a token and always snap to the start or end of it. To avoid this issue, step size of 5 was chosen as the optimal value that performs within 0.25% of step size one and significantly reduces the likelihood of duplicate input sequences.

7.3.4 Training Epochs

As shown in Figure 7, model is significantly improved by increasing the amount of epochs to 50. With this amount of epochs, the model reached a F1 rating of 94.2%, although it should be noted that models trained for only two epochs already reached an average F1 of 90.7%. Further epochs addded insignificant improvements; training a model for 1000 epochs only improved the F1 by 0.5%. However, training time increases linearly with more epochs and too much training causes overfitting. Because of this, 50 was chosen as the optimal point.

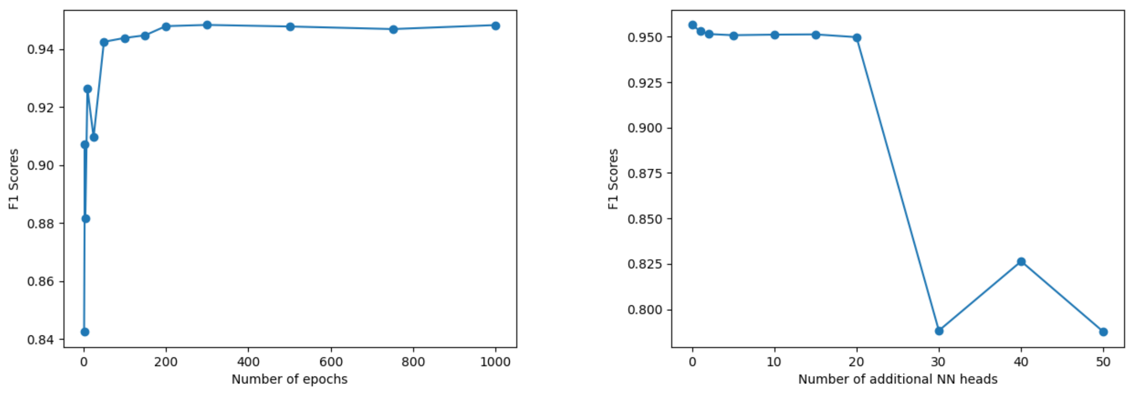


Figure 7: Number of epochs and number of additional NN heads

7.3.5 Number of additional attention heads

Number of additional attention heads determines how many parallel calculations are done within the transformer model. In theory, having more attention heads working in parallel should increase the performance of the model. But in reality, this causes generalization and decreases the accuracy of the model. This is demonstrated on Figure 7. As the number of heads goes from 0 to 20, the performance gradually decreases from 95.3% to 91.5%, but then it drops significantly to 74.8% at 30 heads. This is consistent with findings of Michel et al. [45], where attention heads were pruned during a machine translation task using BERT transformer model and the researchers concluded that in most scenarios, several attention heads can be pruned without noticeable difference. Furthermore, they found that in certain cases, removing attention heads resulted in a better performance (higher BLEU score).
Evidently, having more heads jointly attend multiple positions of the sequence results in worse performance for this task while also increasing the complexity of the model and the training time. Since there is no benefit to having additional attention heads, 0 was determined to be the optimal value.

7.3.6 Embedding Size

The first encoder layer in the transformer architecture embeds each input token into a vector of the embedding size. A curious observation of testing this parameter was that when it was set to a value over 100, sometimes the model would fail to train properly, resulting in a signfiicantly decreased F1 score on the test set, ranging from 15% to 35%. However, when such models did not fail to train, they performed only slightly worse than the models with embedding size of less than 100. The models that consistently performed well and did not fail to train had an embedding size of 64, with other models gradually got worse and the training time also increased as this parameter grew.

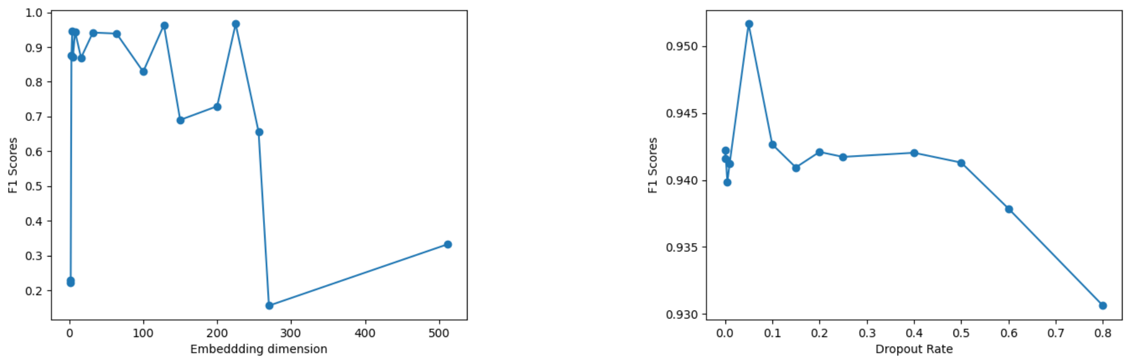


Figure 8: Embedding dimension and dropout rate

7.3.7 Dropout Rate

Dropout is used to 'drop', or in other words ignore some of the input units and detach their connections within the neural network. The premise of this procedure is to prevent the model from overfitting on the data. Dropout has been found to enhance generalization in classification tasks across various domains [46]. Dropout rate refers to the value that determines the probability of each unit to be dropped. The value ranges from 0.0 to 1.0, with 0% and 100% probability respectively. A dropout of 0.1 is used in for the original transformer in [15] and a rate in range 0.1-0.3 is typically used. Figure 8 (right) shows the results of experiments with various dropout rates. Although the transformer models with dropout of 0.1-0.3 performed favorably, the models with a dropout of 0.05 outclassed them, peaking at an F1 score of 95.1%.

Dropout rate	0.0	0.001	0.005	0.01	0.05	0.1	0.15	0.2	0.25	0.4	0.5	0.6	0.8
F1 Score	94.1%	94.2%	93.9%	94.1%	95.1%	94.2%	94.0%	94.2%	94.1%	94.2%	94.1%	93.7%	93.0%

Table 3: Dropout rate

7.3.8 Batch Size

Batch size determines the number of training sequences are put into the model at the same time. The model learns on the batch and then updates the weights. Typically, batch size should not have a very strong impact on the final performance of the model. As shown in Figure 9, there is only a small variance of 0.6% in terms of F1 score between models of batch size 250 and 2000. The general trend seems to be that the larger the batch size, the better the model. However, there is substantial oscillation in the data, which can be explained by the shuffling of the dataset before and during the training process of the models. One downside of using a large batch size, such as 2000, is that it requires linearly more operating memory. On the other hand, the training time decreases as this parameter is higher. Overall, a batch size of 1000 offers a balance of performance, training time and memory constraint.

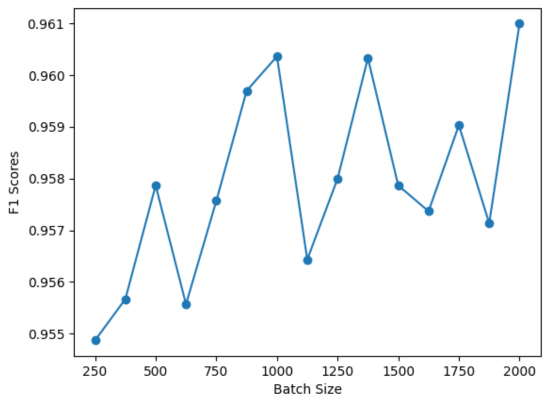


Figure 9: Optimal Batch Size

7.3.9 Optimizer

Upon examining the set of optimizers offered by Keras Framework, obvious outliers are Stochastic Gradient Descent (SGD) and Follow The Regularized Leader (Ftrl) optimizers that operate substantially worse than all the others. This is because the former is developed to convege towards a global optimum, which might not necessarily exist in this scenario, while the latter requires a much diverse feature space to perform well on such a task. All other optimizers are adam-based and perform similarly. RMSprop is the best performing optimizer, but only by a very small margin of 0.3%.

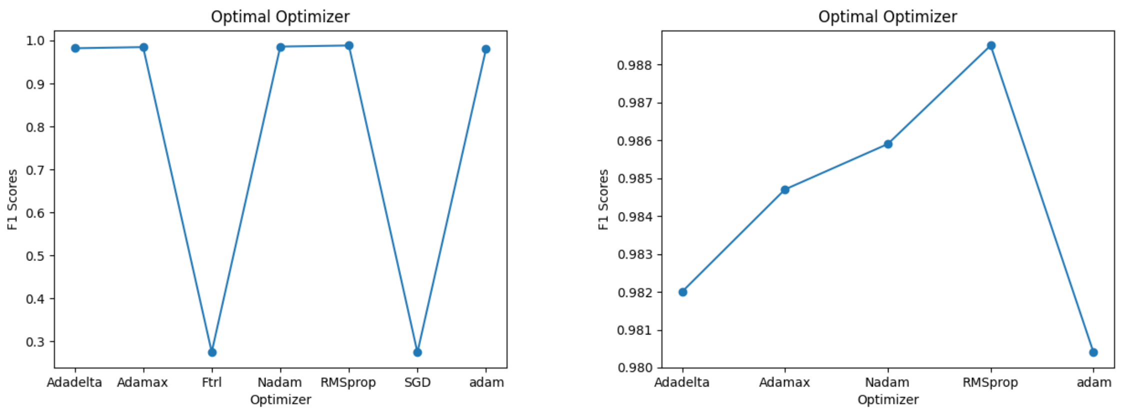


Figure 10: All optimizers (left) and best optimizers (right)

7.3.10 Optimal Transformer Parameters

As a final result of section 7.4, all experiments conducted on individual hyperparameters were taken into account and transformers with optimal values were trained and tested. The optimal parameters for transformer model were found to be:

Parameter	Value
Epochs	50
Batch size	1000
Vocabulary size	20000
Window length	250
Step size	5
Embedding Dim.	64
Num. Additional Heads	0
Dropout	0.05
Optimizer	RMSProp

Table 4: Optimal transformer parameter values

7.4 Performance of optimized transformer models

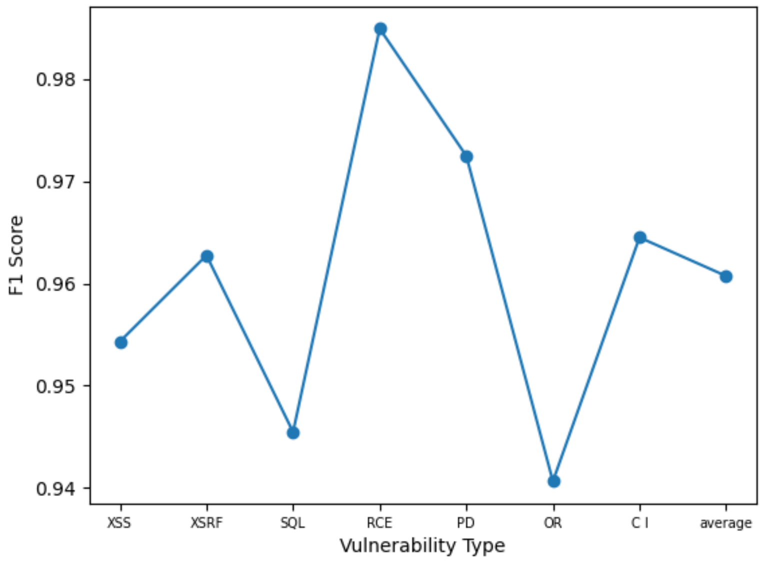


Figure 11: Performance of hyperparameter-optimized transformer

After testing the optimal configuration, results yielded an impressive average of 96.0% F1 score, ranging from 94.5% to 98.4% among different vulnerability types. On average, the optimized transformer models performed 9% better than the Word2Vec-LSTM models from VUDENC. Specifically, these models were far superior in identifying SQL Injection and Remote Code Execution vulnerabilities in code. The least improvement was made on Command Injection subset, but models trained on this subset already had over 90% F1 score in VUDENC.
Transformers performed 19% better than the C2V-LSTM models. In fact, transformers outperformed the character-to-vector LSTM models across all subsets.

7.5 Models trained on multiple vulnerability subsets

Counterintuitively, the models that were trained on 70% of data from each subset of vulnerability performed worse than those that were trained on 70% of the randomly mixed database.
Figure 12 illustrates how well such models (first approach as described in Section 6.5)

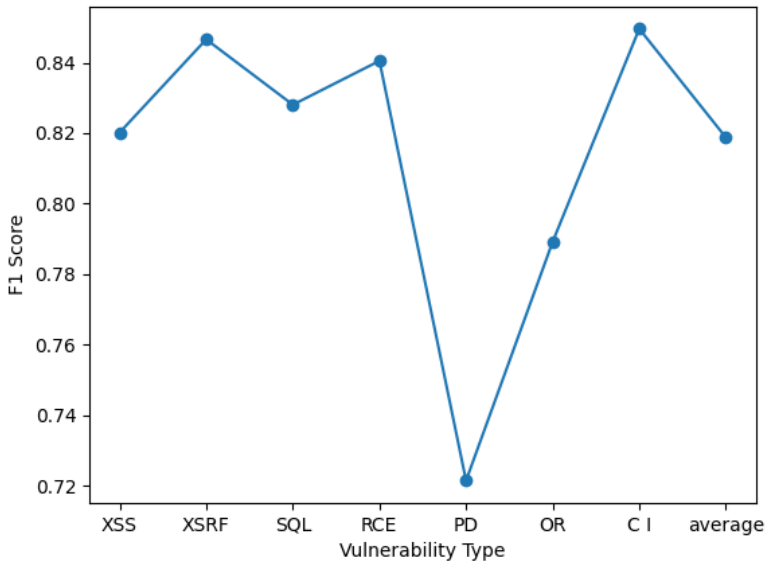


Figure 12: Multi-subset transformer performance

could identify vulnerabilities from individual subsets. As seen in the figure, the average performance of the multi-mode transformer models was about 82.0%. This is a significant loss of precision and recall, with an average of 14.4% loss in F1 score. In particular, the Path Disclosure subset suffered the most losses, dropping from 97.2% to 72.1%. A curious observation from the data is that the F1 scores on individuals subsets relative to each other do not follow the same pattern as with the models trained on specific subsets. For instance, as Figure 11 shows in Section 7.4, Open Redirect vulnerabilities were the

hardest to identify, as opposed to Path Disclosure being the worst here.
The same metric for models trained on aggregated data (second approach as described in section 6.5) that was shuffled first and did not guarantee 70-15-15 split per vulnerability subset was significantly higher at 93.2%. It was pointless to test such models on individual vulnerability types since it could not be identified which parts of the vulnerability subset the model had been trained on.

7.6 Performance of BERT-based Transformer Models

Interestingly, the RoBERTa models outperformed the optimal transformer models but only slightly. Cased versions of the transformers were used, since uncased RoBERTa, BERT and DistilBERT models had lower efficiency than their cased counterparts. This is expected, since in some situations Python token case can help distinguish between internal keywords ('dir') and variable names ('Dir').
Both BERT and DistilBERT models achieved an average F1 score of approximately 97.5%. The relative performance on vulnerability subsets stayed the same on all models. All three of the models have been evaluated on the GLUE benchmark in [36] (BERT), [37] (DistilBERT) and [38] (RoBERTa). The only text classification task on the GLUE benchmark is *The Stanford Sentiment Treebank* (SST-2) that consists of texts from movie reviews and their annotation as positive or negative, where the task of the model is to predict one of the two classes. Notably, all three models performed favourably on this task and showed the best accuracies out of all tasks: 92.7, 91.3 and 92.4 respectively. This suggests that these models are particularly well suited for sentiment analysis, and in extension, text classification tasks. As such, they could be trained to become excellent models for automatically detecting vulnerabilities.

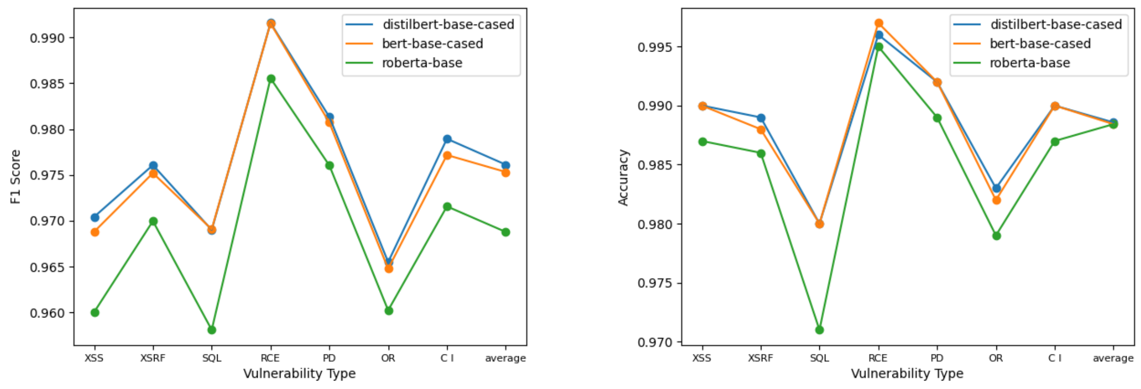


Figure 13: Performance of transformer models implemented by Flair Framework. F1 (left) and Accuracy (right) scores.

7.7 Comparing the results

Average F1 scores of different models implemented in this work is presented in Table 5.
C2V-LSTM: Character-to-vector LSTM.
TR: Transformer with optimal parameters.
MUL-TR: Transformer trained on multiple vulnerability subsets.
AG-TR: Transformer trained on aggregated dataset.
It is clear from the data in table that the Transformer model outperforms both W2V-LSTM (VUDENC) and C2V-LSTM techniques by a wide margin. The best overall F1 score was achieved by DistilBERT model with an impressive 97.6% average.

Vulnerability	VUDENC	C2V-LSTM	TR	MUL-TR	AG-TR	DistilBERT
sql injection	80.1%	63.1%	94.5%	82.8%	N/A	96.9%
xss	86.0%	72.7%	95.4%	82.0%	N/A	97.0%
command injection	90.5%	85.2%	96.4%	84.9%	N/A	97.8%
xsrf	89.0%	81.4%	96.2%	84.6%	N/A	97.6%
remote code execution	88.8%	91.1%	98.4%	84.0%	N/A	99.1%
path disclosure	88.1%	64.9%	97.2%	72.1%	N/A	98.1%
open redirect	87.3%	79.9%	94.0%	78.9%	N/A	96.5%
average	87.1%	76.9%	96.0%	81.8%	93.2%	97.6%

Table 5: Comparison of different models' performances (values given as F1 scores)

8 Discussion and Future Work

This section discusses the experimental results and the evaluation further. It compares the character-level LSTM and transformer-based models with other state-of-the-art approaches. It discusses threats to validity of the evaluation.

RQ1: How effective are C2V and Transformer models at detecting vulnerabilities?
The evaluation showed that while the character-level model achieved an average of approximately 76.9% F1 score, it was still lackluster compared to the original LSTM vulnerability detection tool. Furthermore, character-level models required significantly more training time and memory, since each character had to be embedded into a vector of certain length, multiplying the data size by the vector size. In contrast, transformer models were proven to be much more effective at the given task. Implementing the original transformer and training it on the given database yielded an average accuracy score of almost 99% and an average F1 score of over 96.0%. More complex transformer-based models, such as Google's Bert, were able to further improve the F1 score to over 97.6%.

RQ2: Is it possible to optimize VUDENC and improve its F1 efficiency by applying different and more recent deep learning techniques?
The evaluation showed that although the character-level model performed worse overall, it made small improvements in certain vulnerability types. On the other hand, all transformer models that were used significantly outperformed the LSTM models across the board and detected more vulnerabilities. The average F1 score of the C2V-LSTM model was found to be 76.9%, approximately 10% decrease from the W2V-LSTM model. The same average score for the Transformer model was 96.0%. Most significantly, the average Recall score across the vulnerability types was increased from 84% to 96%, which means that the chance of missing vulnerabilities in the source code decreased by a factor of four. Another improvement when using transformer based models was their training time and memory used. As opposed to a trained embedding, such as word-to-vector or character-to-vector, a basic tokenizer was used to encode sequences into vectors. As such, the input sequences decreased in memory size as they were encoded into vectors and the training required less time. Moreover, unlike LSTM, transformers do not need to process every word individually, rather they process the entire sentence. This also speeds up the procedure of processing all the input data.

RQ3: Is it possible to train a model that can detect multiple vulnerability types? How efficient is such a model?
The evaluation showed that it is possible to train a model that can detect multiple vulnerabilities. Models generated from the first approach achieved an average F1 score of approximately 82% that is substantially lower than that of models trained on individual vulnerability types. However, a score of over 0.8 is still considered to be proficient for NLP tasks. The second approach showed more promising results, with an average score of 93.2%.

8.1 Comparison with other works

There is no direct comparison that can be drawn with the works in the field because of the specific dataset used in this work. However, there are some works that take a similar approach that are discussed in this section.

Thapa et al. [47] used large transformer models on the data published by CGCL/SCTS/BDTS Lab1, Huazhong University of Science and Technology [48], consisting of C/C++ code with two types of vulnerabilities: *Buffer Error* (BE) with 15929 train and 3982 test examples, and *Resource Managment Error* (RME) with 6214 train and 1553 test examples. A third dataset was used that contained a mix of both vulnerabilities with 22072 train and 5518 test examples. The work featured a wide range of models trained on the dataset, such as *Bidirectional Long Short Term Memory* (BiLSTM) and different types of *Transformers*, notably **DistilBERT**, **RoBERTa**, **CodeBERT** , **Megatron-LM** and various version of **GPT** models. After comprehensive testing, Thapa et al. present their findings in a table showing FP, FN, Precision, Recall and F1-score of all models. DistilBERT and RoBERTa models achieved results that mimic the performance in this thesis, with 96.36% and 96.59% F1 score on the Buffer Error dataset. The LSTM-based model (BiLSTM) fell behind in their work there as well, at 76.95%. The best performing models were found to be **GPT-2 Large** at 97.33% and **GPT-2 XL** at 97.31%.

N. Ziems and S. Wu presented their work [52], where they used LSTM, BiLSTM and BERT models to identify vulnerabilities in C/C++ source code. Their data was gathered from the *Software Assurance Reference Dataset* (SARD) database, 50% of the files containing vulnerable code, while the other 50% contains their fixed versions, each marked with one out of the 124 unique labels CWE labels in total (including 'non-vulnerable'). The first approach was to train LSTM, BiLSTM and BERT models individually, resulting in accuracy scores of 72%, 79% and 85% respectively. However, the unique approach of this work was to use BERT in conjunction with LSTM and BiLSTM. Last two experiments with BERT comprised of inputting the output of the model into LSTM and BiLSTM models. The purpose of this is to keep information about the code context of the sequence. These models were able to achieve 93.19% (BERT + LSTM) and 93.49% (BERT + BiLSTM) accuracy scores.

In [49], Alenezi et al. introduce automatic vulnerability prediction with deep neural networks using a fastText n-gram based character level embedding. The data was regarded at slice granularity and contained about 420000 labeled slices with approximately 56000 identified as vulnerable, collected from C/C++ programs at NVD and *Software Assurance Reference Dataset* (SARD). The generated data contained four types of vulnerabilities: *Function Call* (FC), *Array Usage* (AU), *Pointer Usage* (PU) and *Arithmetic Expression* (AE). This approach achieved an average of 98.55% F1-score across all four subsets.

Similar to the approach used in VUDENC, Bagheri et al. [50] scraped GitHub to collect Python sourcecode in order to train LSTM models on three different types of embeddings (word2vec, FastText, BERT). They gathered a dataset with same vulnerability types as in VUDENC, except *Open Redirect*. Unlike in this work, Bagheri et al. did not implement BERT Transformers, instead they only used its encoding capabilities. In fact, they used Microsoft's CodeBERT, which is pre-trained for programming languages, to convert code snippets into vectors that were consequently fed to LSTM models. The work found that there was no significant difference between word2vec and FastText approaches, but the BERT-LSTM model outperformed the other two by a small margin of 1.6% F1 score. The average for the BERT-LSTM model accross all vulnerability subsets was a 87.1%.

8.2 Limitations and threats to validity

Vulnerability Tagging: one important threat to the validity is that the code was tagged as vulnerable or not vulnerable according to the commits that contained certain keywords that hinted at it. There is no way to check whether the code snippets of the commits really contained a vulnerability, without using an external tool. Thus, the models operate under the assumption that all the training data is labeled correctly, which might not necessarily be true.
Issues with Data: the Python source code database that was aggregated from Github is not a comprehensive dataset that exhibits Python code in its entirety. This is because the code of non-commercial, open source projects differ from commercial, internal code that may not be openly published.
Vulnerability Types: data used represents seven common types of vulnerabilities. But it only covers a very small portion of the types of vulnerabilities that are present in software. A much larger database with considerably more types of vulnerabilities would be needed for a model that could identify problems with software in a real world setting. Even then, such a model would be able to detect vulnerabilities that have already appeared in the training set, leaving the vulnerabilities that have no known examples undetected.
Vulnerability Context: sequences that the models were trained on mostly spanned between 200 and 500 characters, covering at most several lines of code. On one hand, this length is sometimes not enough to encompass entire vulnerabilities, particularly ones that are spread over files containing thousands of lines of code. On the other hand, even when a sequence is tagged as vulnerable by the model, it is still unclear which tokens are causing the vulnerability since there are sometimes over fifty tokens in a single sequence.
Evaluation Metrics: this work focused on accuracy, precision, recall, and most of all, on F1 score. However, these are very simple metrics that are not always deemed sufficient to evaluate NLP models. Much more complex metrics, such as the GLUE score, are used to evaluate models on a global scale.
Hyperparameters: although hyperparameters were individually tested and optimized using the baseline model, there is no guarantee that the set of parameter values chosen

for the optimized model are indeed the global optima. Hyperparameter tuning is a very important and complex NLP problem. Various approaches have been proposed for finding the set of best values. But such tools lie beyond the scope of this work. Even so, the parameter data presented in the Experimental Evaluation section demonstrates that setting certain parameter to a sub-optimal value can significantly affect the performance of the model. For example, a transformer embedding size of over 250 sometimes reduced the F1 score of the model to under 20%. Overall, the parameter testing served to avoid such bad cases instead of focusing on finding the absolute best model possible.

8.3 Future Work

The limitations of the study are subject to future research:

- *Vulnerability Tagging, Issues regarding the Data* and *Vulnerability Types* problems could be solved by acquiring a larger learning corpus, with more code and vulnerability types, that was specifically put together to train a vulnerability detection model from sources other than commits on open source projects on github.
- *Vulnerability Context* issue could be solved by training two models: one with a large window length and one with a small one. They could work in conjunction, where the latter homes onto the vulnerabilities detected in the sequence by the former.
- *Evaluation metrics* issue could be solved by simply evaluating the models using more complex metrics
- *Hyperparameters* could be further optimized by applying techniques and toolkits that are designed for this matter

An important question for future research is how the models generated for this work would perform on commercial software and how useful they would be in detecting vulnerabilities. Most of the costs associated with neural network based vulnerability detection tools are about acquiring a large enough training corpus so that the resulting model is efficient at its task. However, it is interesting to investigate how much benefit developers could derive from a tool that tags certain snippets of code as vulnerable.
Another interesting aspect is training vulnerability models on other prevalent programming languages, such as Java, C++, Javascript, C. Although NLP models can be language agnostic, Python code in general is known to be closer to human language than the other programming languages mentioned before. Thus, the performance of NLP models trained on these languages could be less impressive than the one trained on Python. On the other hand, other programming languages, such as COBOL, Alice, Smalltalk, Prolog, are closer to human language and could be more optimal for the text classification task of vulnerability detection. Finally, there is the potential of training a model on multiple programming languages as a solution for vulnerabilities in projects

that have code from different languages.
One other exciting consideration is extending the text classification task with machine translation. An additional model could be trained on tuples each containing vulnerable code and its fixed version. In other words, a model would learn to 'translate' vulnerable code into clean code. This process is semantically and conceptually very similar to spelling, grammar, punctuation checking with suggestions. One example of such a system is Grammarly, which uses transformers to correct mistakes in text [31].

9 Conclusion

This thesis introduced the issue of vulnerability detection and outlined why this issue is important. It described some techniques that are used to identify vulnerabilities and how they developed over the past decades. Background was given about the topics that are crucial to understanding the purpose of the thesis and the realization of the idea. In particular, LSTM and Transformer models were explained, as they are the main underlying architectures used in the implementation. The idea and the theory of the thesis were discussed in Section 5. The technical implementation details were followed in Section 6. After that, the results were presented, evaluated and discussed, followed by ideas for future work.

The systematic experiments that were conducted showed that VUDENC could be improved without using different database; using the transformers implemented via Keras framework, average scores across all vulnerabilities were improved as follows:

- Accuracy from 96.8% to 98.8%
- Precision from 91.6% to 96.1%
- Recall from 83.3% to 96.0%
- F1 from 87.1% to 96.0%

Additionally, more complex transformer-based models were tried to improve the F1 scores further. Three models were tested - BERT, DistilBERT and RoBERTa, with 97.4%, 97.6% and 96.9% average F1 scores respectively. All of them improved upon the base transformer and showed promising results overall. Finally, an attempt was made at training more comprehensive models that would cover multiple types of vulnerabilities. In order to do this, the data for each of the seven vulnerability types was first chunked into train-validation-test parts and then combined into three big chunks as train-validation-test representing all vulnerability types. The models trained and tested on this data yielded an average F1 score of 82.0%.
Second approach did not chunk the individual vulnerability types, but instead combined all of the dataset into a sizable one and split it into train-validation-test parts. Transformer models trained on this data had even better performance with an average of 93.2% F1 score.

References

[1] T. Fawcett, ”An introduction to ROC analysis," Pattern Recognition Letters, Volume 27, Issue 8, 2006, Pages 861-874, ISSN 0167-8655, https://doi.org/10.1016/j.patrec.2005.10.010.

[2] G. Chowdhury, “Natural language processing,” Annual Review of Information Science and Technology, 37. pp. 51-89, 2003, ISSN 0066-4200.

[3] J. Hirschberg, C. D. Manning. "Advances in natural language processing," Science 349, no. 6245, 2015: 261-266.

[4] G. Chowdhury, “Natural language processing,” Annual Review of Information Science and Technology, 37. pp. 51-89, 2003, ISSN 0066-4200

[5] “ShellShock: All you need to know about the Bash Bug vulnerability,” https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=5ee60f4e-030f-4691-b5b4-dc3c9e3701d4&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments

[6] “Cost of a Data Breach Report 2021,” IBM, 2021

[7] Z. Malekos Smith, E. Lostri, J. A. Lewis, “The Hidden Costs of Cybercrime,” 2020.

[8] “Security Primer - EternalBlue”, MS-ISAC, Center for Internet Security, 2021, https://www.cisecurity.org/insights/white-papers/ms-isac-security-primer-eternal-blue

[9] The Common Vulnerabilities and Exposures, https://cve.mitre.org/

[10] B. Chess, G. McGraw, "Static analysis for security," in IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79, Nov.-Dec. 2004, doi: 10.1109/MSP.2004.111.

[11] T. Ball, “The concept of dynamic analysis,” ACM SIGSOFT Software Engineering Notes, Volume 24,Issue 6, pp 216–234, 1999, https://doi.org/10.1145/318774.318944

[12] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," In Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201-213. 2016.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong. "Vuldeepecker: A deep learning-based system for vulnerability detection," 2018, arXiv preprint arXiv:1801.01681

[14] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, L. Grunske, "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase," *Information and Software Technology 144 (2022): 106809.* arXiv:2201.08441

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, 2017.

[16] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” 2019, ArXiv, abs/1810.04805.

[17] V. Sanh, L. Debut, J. Chaumond, T. Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter,” arXiv preprint, arXiv:1910.01108, 2019.

[18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” arXiv preprint, arXiv:1907.11692, 2019.

[19] A. M. Turing, “Computing Machinery and Intelligence,” Mind, Volume LIX, Issue 236, October 1950, Pages 433-460, 1950, https://doi.org/10.1093/mind/LIX.236.433

[20] W. J. Hutchins, “The Georgetown-IBM Experiment Demonstrated in January 1954,” In: Frederking, R.E., Taylor, K.B. (eds) Machine Translation: From Real Users to Research. AMTA 2004. Lecture Notes in Computer Science(), vol 3265. Springer, Berlin, Heidelberg, 2004, https://doi.org/10.1007/978-3-540-30194-3_12

[21] J. Weizenbaum, “ELIZA—a computer program for the study of natural language communication between man and machine,” Communications of the ACM, 1966, 1;9(1):36-45.

[22] GLUE benchmark leaderboard, https://gluebenchmark.com/leaderboard

[23] S. Hochreiter, J. Schmidhuber, “Long Short-term Memory,” Neural computation, 9. 1735-80. 10.1162/neco.1997.9.8.1735, 1997.

[24] https://techcrunch.com/2016/07/15/pat-launches-private-beta-to-help-ai-understand-what-you-say/

[25] O. Vinyals, A. Toshev, S. Bengio, D. Erhan, "Show and tell: A neural image caption generator," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3156-3164, 2015, doi: 10.1109/CVPR.2015.7298935.

[26] W. Feng, N. Guan, Y. Li, X. Zhang, Z. Luo, "Audio visual speech recognition with multimodal recurrent neural networks," 2017 International Joint Conference on Neural Networks (IJCNN), pp. 681-688, 2017, doi: 10.1109/IJCNN.2017.7965918

[27] R. Singh, J. Devlin, “Deep Learning for Program Synthesis,” 2017, https://www.microsoft.com/en-us/research/blog/deep-learning-program-synthesis/?wt.mc_id=MCR_378116_FB1

[28] “Bringing the Magic of Amazon AI and Alexa to Apps on AWS,” 2016, https://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html

[29] R. Fernandez, A. Rendel, B. Ramabhadran, R. Hoory. “Prosody contour prediction with long short-term memory, bi-directional, deep recurrent neural networks,” Interspeech 2014, 2014, DOI:10.21437/Interspeech.2014-445

[30] A Beginner’s Guide to Neural Networks and Deep Learning | Pathmind

[31] K. Omelianchuk, “Grammatical Error Correction: Tag, Not Rewrite”, 2021, https://www.grammarly.com/blog/engineering/gec-tag-not-rewrite/

[32] Checkmarx, https://checkmarx.com/

[33] Gensim Word2Vec Embeddings, https://radimrehurek.com/gensim/models/word2vec.html

[34] Keras Framework, https://keras.io/

[35] The Flair NLP Framework, https://www.informatik.hu-berlin.de/en/forschung-en/gebiete/ml-en/Flair

[36] BERT base model (cased), https://huggingface.co/bert-base-cased

[37] DistilBERT base model (cased), https://huggingface.co/distilbert-base-cased

[38] RoBERTa base model, https://huggingface.co/roberta-base

[39] X. Zhang, J. Zhao, Y. LeCun, “Character-level convolutional networks for text classification,” NIPS’15: Proceedings of the 28th International Conference on Neural Information Processing Systems, Volume 1, Pages 649–657, 2015.

[40] P. Morrison, K. Herzig, B. Murphy, L. Williams, “Challenges with applying vulnerability prediction models,” HotSoS ’15: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, Article No.: 4, Pages 1–9, 2015, https://doi.org/10.1145/2746194.2746198

[41] VUDENC - python corpus for word2vec, https://zenodo.org/record/3559480#.YtlXJ7ZBwuV

[42] VUDENC - datasets for vulnerabilities, https://zenodo.org/record/3559841#.YtlXQLZBwuV

[43] Sage, an open source mathematical software system, https://www.sagemath.org/

[44] "Information technology — Security techniques — Vulnerability disclosure", ISO/IEC 29147:2018(en), 2018. https://www.iso.org/obp/ui/#iso:std:iso-iec:29147:ed-2:v1:en

[45] P. Michel, O. Levy, G. Neubig, "Are Sixteen Heads Really Better than One?", NeurIPS, arXiv:1905.10650, 2019.

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," J. Mach. Learn. Res. 15, 2014.

[47] C. Thapa, S. Jang, M. Ahmed, S. Camtepe, J. Pieprzyk, S. Nepal, "Transformer-Based Language Models for Software Vulnerability Detection: Performance, Model's Security and Platforms", 2022. arXiv:2204.03214

[48] VulDeePecker. Database of "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". https://github.com/CGCL-codes/VulDeePecker.

[49] M. Alenezi, M. Zagane, Y. Javed, "Efficient Deep Features Learning For Vulnerability Detection Using Character N-Gram Embedding," Jordanian Journal of Computers and Information Technology (JJCIT), 2021. 10.5455/jjcit.71-1597824949

[50] A. Bagheri, P. Hegedűs, "A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python", arXiv:2108.02044, 2021.

[51] 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.

[52] Ziems, N. and Wu, S., "Security Vulnerability Detection Using Deep Learning Natural Language Processing", IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), arXiv:2105.02388, 2021.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den August 19, 2022 [illegible]