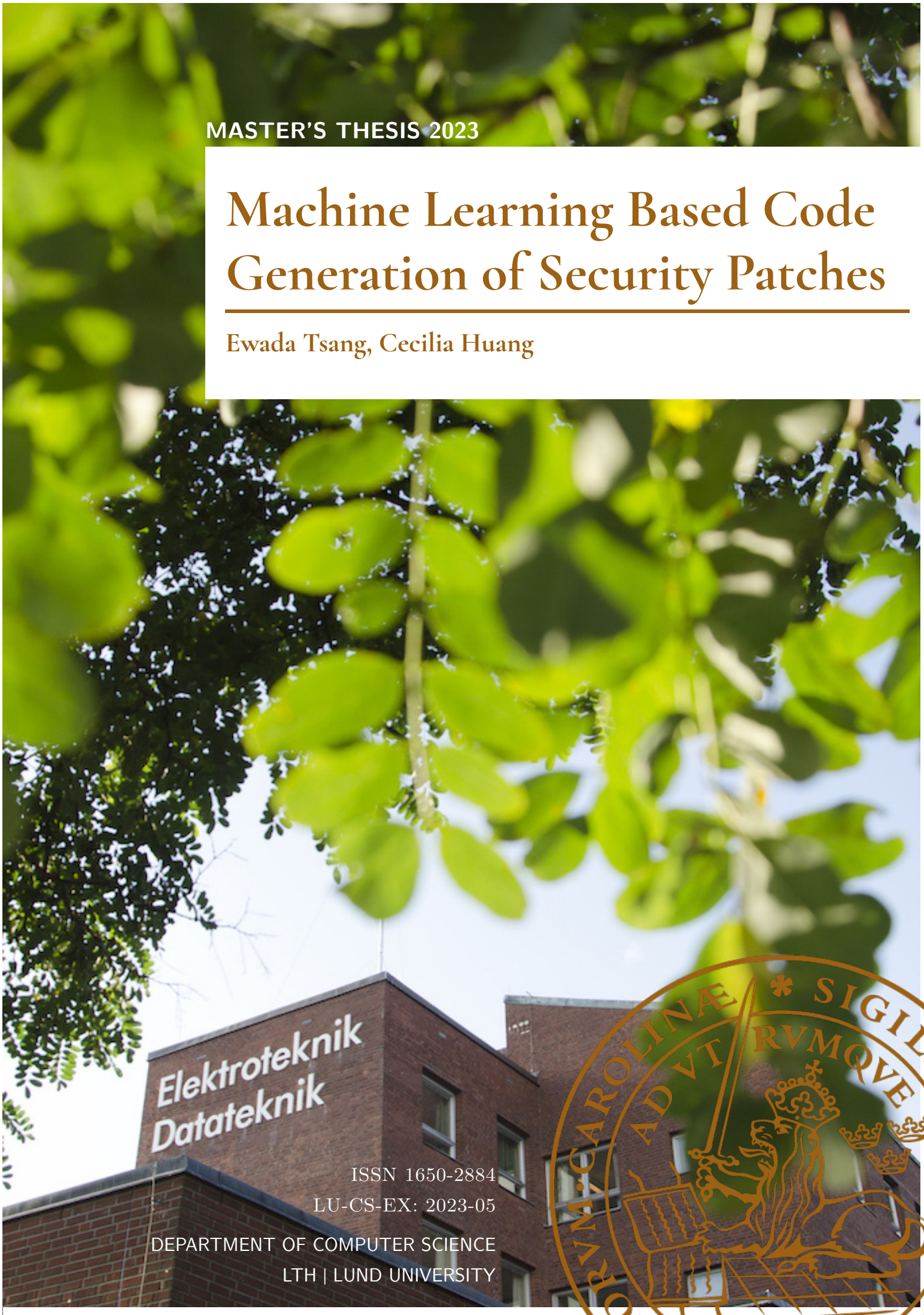

MASTER'S THESIS 2023

Machine Learning Based Code Generation of Security Patches

Ewada Tsang, Cecilia Huang

ISSN 1650-2884
LU-CS-EX: 2023-05

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-05

Machine Learning Based Code Generation of Security Patches

Maskininlärningsbaserad kodgenerering av säkerhetsfixar

Ewada Tsang, Cecilia Huang

Machine Learning Based Code Generation of Security Patches

Ewada Tsang
ew8283ts-s@student.lu.se

Cecilia Huang
ce0263hu-s@student.lu.se

March 27, 2023

Master's thesis work carried out at Debricked a Micro Focus Company.

Supervisors:
Emil Wåreus, emil.wareus@microfocus.com
Noric Couderc, noric.couderc@cs.lth.se
Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Examiner: Pierre Nugues, pierre.nugues@cs.lth.se

Abstract

In this thesis, we explore the feasibility of using a machine learning model to generate security fixes for software vulnerabilities. Patching security bugs is crucial to ensure the safety of both individuals and organizations from malicious attacks, but it is also a time-consuming task. Despite recent advances in machine learning, automatic detection and fixing of security vulnerabilities are still largely unsolved problems. There is ongoing research in this field, which suggests to the potential of using machine learning as a tool to generate security fixes. However, these studies do not address the various variables that influence model performance.

Therefore, we implemented a machine learning model to study how the dataset used to train it affects performance, aiming to understand important factors for practical use. Our observations on the impact of dataset size, lines changed, splitting method, weakness categories, and repository distributions provide valuable insights for developing a more generalizable model and performing data engineering on future datasets.

Keywords: machine learning, security patches, code generation, open-source vulnerabilities

Acknowledgements

Our sincere thanks go to our supervisors, Emil Wåreus, Noric Couderc and Christoph Reichenbach, for their guidance and support throughout our Master's Thesis.

Emil provided daily mentorship and unique insights, while Noric and Christoph kept us on track, offered valuable feedback and motivated us to continue our investigations.

We also want to thank the Data Science team at Debricked, who constantly encouraged us. From day one, you made us feel like part of the team. We give a special shout-out to Filip Hedén, who always went above and beyond to assist us with any challenges we faced.

Contents

Chapter 1
Introduction

As our society continues its shift towards digital technologies, software systems have become increasingly critical to our daily lives. However, this reliance has also resulted in a surge of software vulnerabilities. If left unaddressed, these vulnerabilities have the potential to cause severe security breaches, data loss, and other negative consequences [1].

Manually patching these vulnerabilities is a time-consuming and error-prone process. With software systems becoming more complex and interconnected, the number of vulnerabilities discovered and the effort required to patch them has increased significantly. In fact, according to the National Vulnerability Database [2], the number of reported vulnerabilities has risen steadily over the past decade, with a 24.6% increase in 2022 compared to the previous year, as can be seen in Figure 1.1.

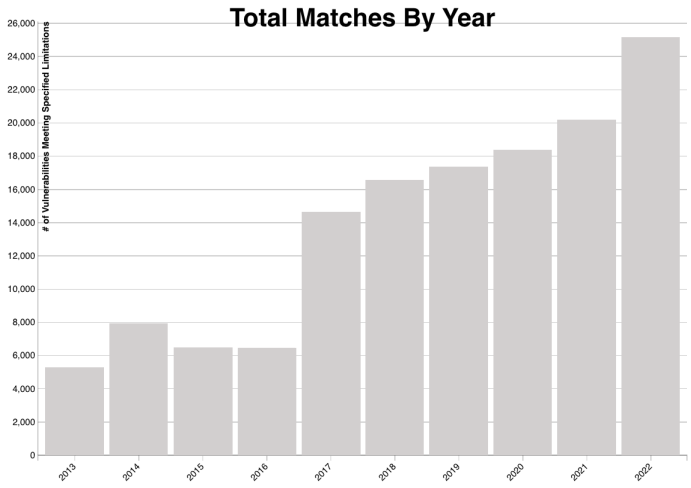

Figure 1.1: Number of reported vulnerabilities per year

The field of software security has faced the challenge of detecting and correcting vulnerabilities in software systems for a long time. While this issue was traditionally addressed

by security experts manually, it has become increasingly difficult to manage due to the scale and diversity of threats. Furthermore, hand-written program repair rules may not efficiently handle new and unknown threats, making it necessary to explore alternative solutions.

To tackle this challenge, researchers have proposed various techniques, including static program analysis and automated program repair (APR). However, automating security vulnerability fixes remains challenging. This topic is not only of great interest for academia, but many companies also addresses this issue. Among these companies is Debricked, a specialized open source software security provider that emerged from a research project from Lund University in 2018. Debricked's mission is to identify, fix, and prevent vulnerabilities in open source dependencies, as well as ensure license compliance and community health [3]. Therefore, it was a great privilege to collaborate with Debricked on this thesis, with the aim of developing a proof of concept using machine learning to generate security patches.

Machine learning based approaches have emerged as a potential solution because they can learn from a large amount of data and generalize to new cases. Recent advancements in machine learning such as the transformer architecture, have created new possibilities for automated program repair using machine learning. Models like GitHub Copilot [4], Salesforce's CodeGen [5], and CodeBERT [6] are trained on programming languages and have demonstrated great potential in the area of generating code.

Although there has been some research on machine learning-driven security patches, it is still a relatively new field. During our literature research, we came across two recent studies that were particularly fascinating: VulRepair proposed by Fu et al. [7] and a study by Huang et al [8]. To the best of our knowledge, there were no other studies in this specific area at the time of writing. These studies served as sources of inspiration and guidance for our research and they have significantly influenced our study.

1.1 VulRepair: A T5-Based Automated Software Vulnerability Repair

VulRepair: A T5-Based Automated Software Vulnerability Repair [7] presents an approach for automated software vulnerability repair using the T5 language model. One of the main contributions of this work is the comparison against VRepair [9], a previous approach that was trained on a small bug-fix corpus of 23,607 C/C++ functions using a Vanilla Transformer.

The researches of VulRepair, Fu et al. [7] argue that by using a pre-trained model, such as CodeT5, VulRepair is able to achieve better results and address the limitations of VRepair. The developers use a dataset of 8,482 vulnerability fixes from CVEfixes and BigVul dataset, from 1,754 real-world software projects to train and evaluate the model. The dataset is randomly divided into 70% training, 10% validation and 20% test. The model uses beam search with a beam size of 50. The security vulnerabilities that VulRepair attempts to patch is C/C++ code, and they do not seem to limit to any number of lines changed or CWE-IDs.

The results of the experiments show that VulRepair has an Exact Match score of 44% and is able to repair 745 out of 1,706 vulnerabilities. The article presents a promising approach for automated software vulnerability repair using pre-trained language models and provides evidence that using a pre-trained model can improve the performance of vulnerability repair systems.

1.2 Repairing Security Vulnerabilities Using Pre-trained Programming Language Models

In *Repairing Security Vulnerabilities Using Pre-trained Programming Language Models* by Huang et al. [8], the developers propose using pre-trained BERT-style language models [10] to automatically repair security vulnerabilities in C/C++ code. The study compares the performance of their method to state-of-the-art Automated Repair (APR) tools such as DLFix [11], SequenceR [12], and CoCoNut [13]. They use the Juliet C/C++ test suite from the Software Assurance Reference Dataset (SARD) as the training dataset, with a focus on five specific weaknesses:

- CWE-121: Stack-based Buffer Overflow
- CWE-190: Integer Overflow or Wraparound
- CWE-369: Divide By Zero
- CWE-401: Missing Release of Memory after Effective Lifetime
- CWE-457: Use of Uninitialized Variable

There are in total 11,221 entries for both single-line and multi-line repairs. The BLEU score is used as the evaluation metric during model training and the model's accuracy is determined by its ability to generate an exact match that exists in the target set. The study achieved an EM score of 95.47% for single-line repairs, which was comparable to state-of-the-art APR tools, and 90.06% for multi-line repairs, which outperformed state-of-the-art APR tools.

Huang et al. [8] does however note that the models have limitations, such as a tendency to generate fixes with syntactic structural integrity errors and poor predictive power when the code sequences are excessively long. Despite these limitations, the methods presented in the article and the previously mentioned VulRepair in Section 1.1 demonstrate promising results and suggest that using pre-trained language models trained on programming languages could be a highly effective approach for repairing security vulnerabilities.

1.3 Aim

The purpose of this report is to explore the applicability of using code-generating models for automated repair of security vulnerabilities. The main objective is to train machine learning models and evaluate how their performance is impacted by different training datasets in order to provide insights for further research in this area. The research questions that we aim to answer are:

- **RQ1:** What are the key influencing factors that impact the model's performance?
- **RQ2:** How does the performance of these models compare to current state-of-the-art approaches in the field?

Chapter 2
Theory

This chapter introduces the concepts of the transformers architecture which is the underlying architecture of the language model we used for this thesis, to provide explaination of why we used the CodeT5 model to train our data. Followed by introductions to automated program repair and security vulnerabilities to provide context for our research questions.

2.1 Transformer

The Transformer is a neural network architecture proposed by Google researchers in 2017 in the paper *Attention is All You Need* [14]. It introduced self-attention mechanisms to weigh the importance of input elements when processing a sequence, enabling the model to learn long-range dependencies effectively, making it highly suitable for Natural Language Processing tasks. The Transformer model has an encoder and decoder part, relying on positional encoding, attention, and self-attention to process input sequences.

Unlike traditional architectures such as Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTMs), and Gated Recurrent Neural Networks (GRUs), the self-attention mechanism is parallelizable, allowing for faster training and inference. Moreover, traditional architectures can only capture context from a limited number of previous time steps, which limits their ability to handle longer-range dependencies.

2.1.1 Encoder

As mentioned before, the transformer architecture proposed in the paper *Attention is All You Need* [14] has two main components, the encoder part and the decoder part. The encoder is composed of a series of identical layers. Each layer consists of two layers in turn: a self-attention layer and a feed-forward network.

The self-attention layer allows the encoder to capture contextual meaning and relationship between the token of the input sequence by computing a weighted sum of all tokens,

where the weight assigned to each token is determined by its relevance to the rest of the sequence. This allows the encoder to capture dependencies between the input elements even in long sequences.

The feed-forward network is used to apply a non-linear transformation to each element of the input sequence independently, which allows the encoder to capture complex relationships between the elements of the input sequence. At each layer, the encoder takes in a sequence of hidden states from the previous layer and passes it through the self-attention mechanism and feed-forward network. The output of each layer is a new sequence of hidden states that capture increasingly complex relationships between the input elements.

Before being processed by the encoder layers, the input sequence is embedded into a continuous vector space using an embedding matrix. The embedding matrix allows the model to represent each element of the input sequence as a dense vector that captures its meaning and context [14].

BERT [10] is one example of transformer-base model that only consist of a stack of encoders. CodeBERT [6] is variant of BERT that is pretrained on a large corpus of source code which makes it specialise in programming languages. CodeBERT is one of the language models Huang et al. [8] used in their research.

2.1.2 Decoder

In an encoder-decoder architecture, the output from the encoder will be fed into the decoder. While the encoder is designed to capture complex relationships between the elements of the input sequence by using a self-attention layer and feed-forward network, the decoder is designed to generate an output sequence that is conditioned on the context of the input sequence and does so by using a self-attention layer, a feed-forward network, and an additional attention layer that attends to the encoder output, which will allow the decoder to take into account the context of the input sequence as it generates the output sequence.

Like the encoder, the decoder also consists of a series of identical layers that in turn consist of these three layers mentioned above. At each time step of the decoder, the output of the previous layer is used as input to the current layer. The decoder generates the output sequence in an auto-regressive manner, where at each step, it predicts the next element of the output sequence based on the previous predicted elements [14].

Examples of decoder models include GPT [15], GPT-2 [16], GPT-3 [17], and GPT-4 [18], which have a stack of decoders without encoders in their architecture.

2.1.3 Encoder-Decoder models: T5 and CodeT5

Having examined the functionalities and applications of both encoder-only models, such as BERT, and decoder-only models, such as GPT, we now shift our focus towards the encoder-decoder models which combine the strengths of both models. As a reminder, the encoder is responsible for processing the input text and producing a latent representation and the decoder generates the output text. Two examples of encoder-decoder models are T5 [19] and CodeT5 [20].

T5 [19] is a large language model that uses both encoders and decoders to perform various NLP tasks and CodeT5 [20] is a fine-tuned version of the T5 model, which is the model used by Fu et al. [7] in their paper. CodeT5 is pre-trained to derive generic representations

for Programming Language (PL) and Natural Language (NL) by using both unimodal data (PL-only) and bimodal data (NL-PL pairs), and uses token type information from code structure to comprehend code semantics. It is designed to generate and understand code in six programming languages: Python, Java, JavaScript, PHP, Ruby, Go, C, and C#. After being pre-trained, CodeT5 was fine-tuned on tasks such as code summarization, code generation, code translation, and code refinement and achieved state-of-the-art performance on various benchmarks [20].

Given CodeT5's high performance in code generation tasks and inspired by the research done by Fu et al. [7], we also decided to use the CodeT5 model for this thesis.

Beam search

Beam search [21] is a technique used by T5 [19] and CodeT5 [20] to generate the most likely sequence of words for a given input. It tracks the top-k most probable sequences of words at each step by selecting k words with the highest probabilities and generating k new sequences. This process is repeated until a complete sequence is generated. While it helps the model find better sequences, beam search can be slow and cannot guarantee the absolute best sequence of words. Nonetheless, it is a valuable technique for generating high-quality sequences [21].

2.1.4 Evaluation Metrics

We chose Exact Match (EM) and Bilingual Evaluation Understudy (BLEU) as metrics to evaluate the models, inspired by recent research papers by Fu et al. [7] and Huang et al. [8]. This section will explain the chosen metrics.

Exact Match Score

Exact Match Score, or Perfect Match Score, is a metric used by both Fu et al. [7] and Huang et al. [8] in their studies to evaluate whether the generated patches match the target patch perfectly. However, it has limitations as it does not account for the similarity between the two texts. The EM Score is calculated using the following formula:

$$\text{EM} = \frac{\text{number of predictions that exactly match the target}}{\text{total number of predictions}}$$

BLEU Score

The BLEU Score is a widely used evaluation metric in NLP, particularly for machine translation and summarization tasks[22]. It measures the similarity between generated and reference texts by comparing their n-grams and ranges from 0 to 1, where higher scores indicate higher similarity. However, a lower score may be acceptable in some cases, while a higher score does not always guarantee accuracy [23]. Generally, a BLEU score of 0.6-0.7 is considered very good [22], and a score of 1 is unlikely to be achieved even by a human translator [23]. The BLEU score as explained by Papineni et al. [23] is calculated as follows:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

- BP: the brevity penalty factor, which adjusts the BLEU score based on the length of the generated translation relative to the reference translations. The brevity penalty factor is calculated as follows:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

  - c is the length of the generated translation.
  - r is the length of the reference translation that is closest in length to the generated translation.

- N is the maximum n-gram order considered. For this study, N is 4 as per the Huggingface implementation [24].

- p_n is the precision of n-grams, which is the ratio of the number of n-grams in the generated translation that appear in any of the reference translations, to the total number of n-grams in the generated translation ($p_n = \frac{\text{Number of correct predicted n-grams}}{\text{Number of total predicted n-grams}}$).

- w_n is the weight for the n-gram precision score, which is typically set to uniform weights ($w_n = \frac{1}{N}$).

To evaluate the quality of the generated security patches, the EM score will be the primary metric with which we will analyze the results, while the BLEU score will be used as a supplementary metric to assess similarity to the target. It is important to note that these metrics will only be compared against the target and not evaluated for their security patching abilities. By using both metrics together, we can provide a more comprehensive evaluation of the security patches.

2.2 Automated Program Repair (APR)

Automated Program Repair has been researched for the past decade. There are many APR tools on the market including Angelix [25] and SOSRepair [26], which are based on semantic analysis and kGenProg [27], a genetic programming approach. However, recent advances in deep learning have contributed to the rise in popularity for data-driven machine learning based software repairs. This thesis focuses on Automated Program Repair from a machine learning perspective.

2.3 Security Vulnerabilities & Weaknesses

Security vulnerabilities are weaknesses in software that attackers can exploit to gain unauthorized access, steal data, or harm systems. The National Institute of Standards and Technology (NIST) maintains a list of publicly disclosed vulnerabilities in the National Vulnerability Database (NVD), which assigns each vulnerability a unique Common Vulnerabilities and Exposures (CVE) number and a Common Weakness Enumeration (CWE) classification. The CWE hierarchy (Appendix A Figure A.1) categorizes weaknesses in software security. This project specifically focuses on a subset of the CWEs that cover the most common weaknesses found in CVEfixes for Python and Java, [28] which will be introduced in the subsequent section.

2.3.1 CVEfixes

CVEfixes is an automated tool that efficiently manages newly discovered or patched vulnerabilities by retrieving JSON feeds from the NVD server and adding annotations with Common Weakness Enumeration and other relevant meta-information. It matches entries on commit, file, and method-level, allowing for convenient querying of the methods before and after vulnerability fix, compared to other databases where one has to find commits through URL's to get hold of the code. The CVEfixes dataset that we have downloaded includes all published CVEs up to 27 August 2022, covering 7,636 CVEs and a total of 7,798 vulnerability fixing commits [29]. The entity-relationship diagram of the CVEfixes database is presented in Figure 2.1 and the thick black arrows represent the specific information that we needed. Throughout the report, this collection will be referred to as CVEfixes, CVEfixes dataset, CVEfixes database or similar.

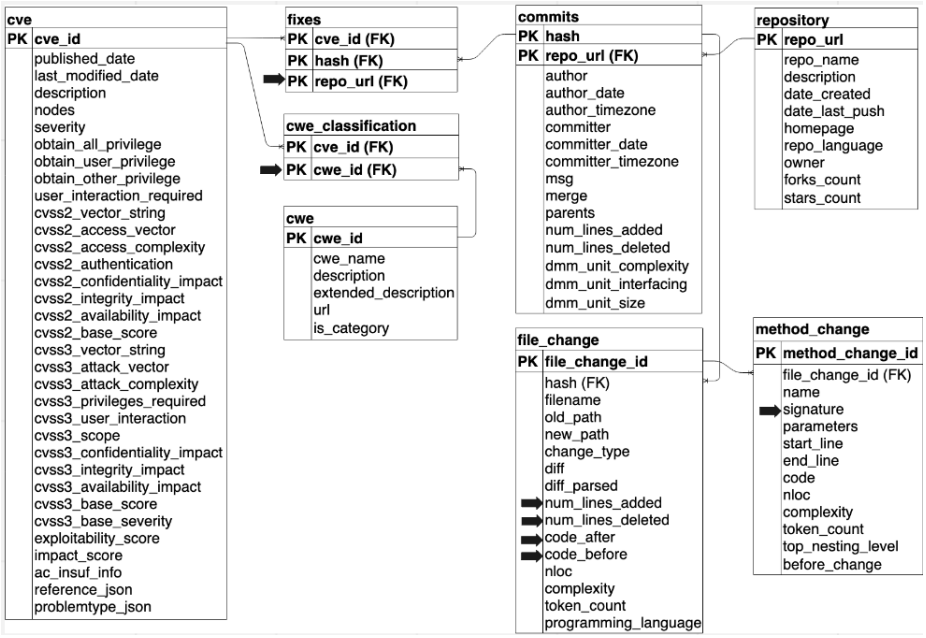

Figure 2.1: CVEfixes Entity-Relationship Diagram by Bhandari et al. [28]. The inserted thick black arrows are our own modifications.

2.3.2 Example

To showcase an example retrieved from CVEfixes [28], we will provide details of what a CVE-entry looks like (CVE-2021-31245 [30]) in Table 2.1. The corresponding code with the vulnerability can be found in Listing 3.1 and its potential fix is in Listing 3.2.

CVE-number	CVE-2021-31245
Description	omr-admin.py in openmptcprouter-vps-admin 0.57.3 and earlier compares the user provided password with the original password in a length dependent manner, which allows remote attackers to guess the password via a timing attack.
Weakness Enumeration	CWE-287 Improper Authentication

Table 2.1: Example of the information from a CVE entry

2.3.3 Open Source Software

Open Source Software (OSS) refers to software that comes with its source code available for anyone to view, use, and modify, as per the Open Source Initiative's (OSI) definition [31]. This means that the code is open for collaboration and improvement, and aims to increase the accessibility and transparency of software. It is worth noting that the CVEfixes database only contains open-source software. Closed-source, or proprietary software contains source code that is not accessible to the general public.

The difference between open-source and closed-source software extends beyond just the accessibility of the code. In terms of software security, the two differ in patch behavior as well. Open-source vendors have been found to release security patches 3-10 times faster than proprietary vendors [32].

Potential explanations to this could be because open-source software is generally more exposed to attackers, making it more susceptible to bugs and vulnerabilities, which are then reported faster. Additionally, with more eyes on the code due to the open nature of OSS, more people may be able to find and fix security issues. On the other hand, closed-source software is harder for attackers to access, making it slower to attack. However, the limited access to code also means there are fewer people to find vulnerabilities and fewer individuals to fix them.

Chapter 3
Method

The following section provides a detailed description on the process from Data Engineering to fine-tuning the model, as well as some dataset insights.

3.1 Data Engineering

When training a machine learning model, the most important step is to obtain relevant data for the task at hand. We started our search by exploring various potential datasets, such as SARD [33], which was utilized in Huang et al.'s research [8], and CVEfixes [28], which was employed in VulRepair's [7] study. After thorough evaluation, we concluded that the CVEfixes dataset was the optimal choice based on the following factors:

- The dataset contains actual commits rather than synthesized data, making it a preferred option over SARD's test suites.
- The data is easily available and neatly organized in a SQLite3 database, which made the query process straightforward using SQL.
- The dataset have a variety of useful metadata, such as code before and after commits, commit URLs, CWE-IDs, and method signatures.
- CVEfixes is a collection tool, which means that more data can be gathered at any point in time.

3.1.1 Exploratory experiments

During the initial stages of our project, we needed to become familiar with the CVEfixes dataset. To accomplish this, we set up a quick training loop in Google Colab that allowed us to iterate rapidly and identify any limitations we would need to consider. We conducted

a series of experiments to evaluate the impact of various factors such as the number of lines changed in each file, programming languages, and CWE-IDs. These experiments served as the foundation for our design decisions in the project.

After analyzing the results of our experiments, we discovered that our model performed best when the number of lines added and deleted in each file was no more than 5. As a result, we chose to use 5 lines added and deleted as one dataset. To increase the number of data entries, we created another dataset with lines added and deleted of up to 15. Additionally, we reviewed the dataset and selected the 5 most common CWE-IDs to use in our model. We chose this number because a related study by Huang et al. [8] also used 5 CWE-IDs. However, this approach limited our dataset. To address this, we decided to include other CWE-IDs within the same category, as shown in Appendix A, Figure A.1. This decision was based on guidance from Debricked, which suggested that including related CWE-IDs could potentially enhance our model's performance.

Target Language

When deciding on the target language for the project, the six languages that CodeT5 is pre-trained on was evaluated and ultimately, Python and Java were selected due to their relevance to Debricked's business needs, as well as our familiarity with the languages. This allowed for manual inspection of the generated output from the model, which was particularly useful for evaluating the quality of the data. Additionally, a mixed dataset comprising both Java and Python data were incorporated for two main reasons. Firstly, we hoped that augmenting the dataset with two languages would help improve the performance of the model by increasing the number of entries. Secondly, we were interested in exploring how well the model could handle a mixed language dataset. This mixed dataset will be referred to as Mixed throughout the report. After our exploration phase, we proceeded to finalize the datasets that were going to be used for training, as detailed in the following sections.

3.1.2 Finalized training datasets

The final datasets used to train the models in this study were obtained from the CVEfixes dataset, as previously mentioned, with design choices motivated by our exploratory experiments. Each data entry contains the following information:

- Vulnerable code: the code before the commit.
- Fixed code: the code after the commit.
- The CWE-ID connected to the commit
- The signature of the vulnerable method
- The year of the commit
- The URL address to the commit

3.1.3 Dataset parsing, special tokens and splitting

After the data is queried, the data processing starts. The program we implemented for this project will perform the following steps:

1. There are five CWE-folders each for Python and Java. Inside each language folder, there are source-code files with code before fix and code after fix.

2. Each source code file is then parsed and the vulnerable method that matches with the signature gets extracted.

3. After the method is extracted, whitespaces, single-line and multi-line comments are removed.

4. A DataFrame is then created with all the relevant information contained in columns. All code duplicates are removed and special tokens are inserted. We have provided some examples in Listings 3.1 and 3.2 that showcase what the data looks like. The localization for where there have been code changes are marked with:

   <S2SV_StartBug> vulnerable line **<S2SV_Endbug>**

   The vulnerability fix will be inside the tokens:

   <S2SV_StartBug> fixed line **<S2SV_Endbug>**

   There are three possible cases when special tokens are added and methods in the dataset can consist of a combination of these:

   (a) **Modification:** The changed line(s) gets added with special tokens in 'vul' and the 'fix' consists of segments of vulnerability fixes. If there are multiple modifications the 'fix' will look like:

   <S2SV_ModStart> fix 1 **<S2SV_ModEnd>** **<S2SV_ModStart>** fix 2 **<S2SV_ModEnd>** ... **<S2SV_ModStart>** fix n **<S2SV_ModEnd>**

   (b) **Code added:** If there is only new line(s) of code added, 'fix' will contain the new code (same as Listing 3.2 for single line and the above example for multi-lines) and 'vul' will consist of empty token(s):

   <S2SV_StartBug> **<S2SV_EndBug>**

   (c) **Code removed:** If the 'fix' has removed line(s) of code, then it will consist of empty token(s):

   <S2SV_ModStart> **<S2SV_ModEnd>**

   While our implementation adheres to the state-of-the-art standards and draws inspiration from the data processing methodology utilized by both Fu et al. [7] [34] and Chen et al. [9], we will be deviating from Fu et al.'s methodology going forward. Specifically, we will be using different split ratios and methods and stratifying the CWE-categories. More details on these changes will be provided in the next point.

5. The dataset up to this point is still separated by CWE category and after the special tokens are added, the datasets gets split into training, validation and testing sets while still being in respective categories. We thereby split each category using both a Random split where the data is shuffled, and a Time split, where the data is ordered by chronological order. Hence, for the Time split, the training/validation data will contain older entries and the validation/test will have the newest data.

   The reason for using two splits is to determine the impact of time on model performance and to investigate whether the data changes over time, which would be crucial to consider when building models for real-life applications. It is important to note that the time split is based solely on the commit year, meaning that the entries in the training, validation, and test sets are not strictly arranged by chronological order. This is because some years can overlap in the data, leading to a lack of strict temporal ordering. Additionally, we want to be clear that Random split allows data from the same commits, however, we have focused on ensuring that there are no duplicated methods within the dataset splits or across the splits.

6. The full dataset is first divided into 80% training and 20% test. The training dataset is further split into 80% training and 20% validation, resulting in 64%/16%/20% for training/validation/test. While the ideal splitting ratio may vary depending on the specific context, we acknowledge that the splitting ratio we used may not have been optimal for our model. However, we were able to successfully train our models with this split and obtain meaningful results for analysis, so we decided to use this ratio in our study.

7. After each CWE category has been split, we merge the respective splits so that we in the end have training, validation and test splits that are stratified, meaning that the CWEs within each split is kept at a similar ratio. Figure 3.1 shows the splitting and merging process.

```
CWE-287
def verify_password(plain_password, user_password):
    if <S2SV_StartBug> plain_password == user_password: <S2SV_EndBug>
        LOG.debug("password true")
        return True
    return False
```

Listing 3.1: Example: vulnerable method in Python with CWE-ID and special tokens.

```
<S2SV_ModStart>
secrets.compare_digest(plain_password,user_password):
<S2SV_ModEnd>
```

Listing 3.2: Example: the vulnerability fix contained within the special tokens.

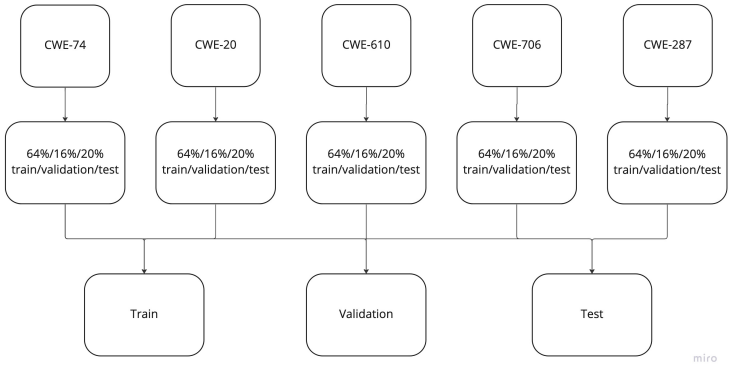

Figure 3.1: Illustration of the splitting process. In the end all training, validation and test sets get merged.

3.2 Hyperparameter tuning

After data processing, our models are hyperparameter-tuned in Vertex AI with automated search. Vertex AI is a Google Cloud platform specifically for building, training, and deploying machine learning models. We narrowed down the search space for hyperparameters by setting ranges based on promising results from the exploratory phase and conducted 10 trials to fine-tune the model using different sets of values. Our hardware configuration utilized a2-highgpu-1g as the machine type and NVIDIA TESLA A100 as the accelerator type. We present the results of our hyperparameter tuning in Table 3.1.

To fine-tune the model, a standard machine learning training loop is employed, which incorporates the AdamW optimizer (an extension of Adam optimizer with weight decay regularization to mitigate overfitting) and a linear learning rate scheduler with warmup. We save the model checkpoint each time the validation loss improves, and then evaluate them using our test set. In the test loop, the model is configured to generate 30 beams, and the raw predictions are saved to file. We then go through the hyperparameter tuning trials and select the models with the highest performance for each dataset.

Models		**Hyperparameters**					
Language	Split	Lines Changed Restriction	Epochs	Batch Size	Learning rate	Weight Decay	Hyperparameter Tuning Duration
Python	Random	5	10	8	4E-4	0	1h 52min
		15	15	8	3E-4	1E-2	2h 3min
	Time	5	10	4	5E-4	1E-3	1h 55min
		15	10	6	5E-4	1E-2	1h 58min
Java	Random	5	10	6	3E-4	1E-3	1h 53min
		15	15	6	5E-4	1E-3	1h 52min
	Time	5	15	6	7E-4	0	1h 51min
		15	15	6	5E-4	1E-2	1h 51min
Mixed	Random	5	11	6	1E-4	4E-4	1h 51min
		15	10	6	3E-4	1E-3	2h 20min
	Time	5	15	6	4E-4	1E-3	1h 56min
		15	10	8	5E-4	1E-2	2h 19min

Table 3.1: Hyperparameter tuning results for each dataset with Python, Java or Mixed, Random or Time-based split and 5 or 15 lines addition and deletion restrictions.

3.3 Dataset Size

This section provides an overview of how the size of the our datasets was impacted by the preprocessing and data engineering steps. We describe the number of entries after processing, the impact of limiting CWE categories and the number of entries per language.

3.3.1 Number of Entries After Processing

In Section 3.1.3, we processed datasets by extracting vulnerable methods and removing duplicates. Table 3.2 shows the decrease in entries for each dataset after processing, ranging from 16.1% to 35.8%. Python still has the largest datasets, with around 32% more entries in the 5 Lines and 15 Lines datasets compared to Java.

Language	Lines Restriction	Before processing	After processing	Decrease	Decrease (%)
Java	5 Lines	156	111	45	28.5
	15 Lines	310	199	111	35.8
Python	5 Lines	174	146	28	16.1
	15 Lines	330	262	68	20.6

Table 3.2: Number of entries lost after data processing.

3.3.2 Number of Entries After Limiting CWEs

We limited the number of CWEs to five categories to improve model performance, but this decision also reduced the number of available entries. Figure 3.2 shows that the 5 Line and 15 Line datasets lost 28.2% and 41.9% of entries, respectively, resulting in 257 and 461 entries after the restriction.

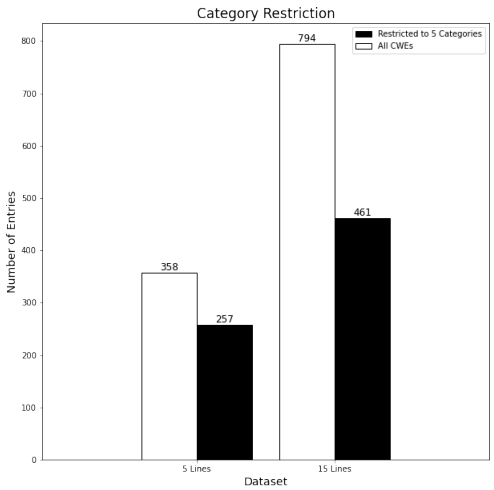

Figure 3.2: Number of entries in 5 Line and 15 Line datasets after restricting to five CWEs categories

3.3.3 Number of Entries per Language

To boost dataset size, we created a Mixed dataset consisting of both Python and Java entries. We also further created a dataset with 15 lines of change, which increased the dataset with about 80% for each language respectively. Table 3.3 summarizes the number of entries for each dataset.

Language	5 Lines	15 Lines	Increase	Increase (%)
Java	111	199	88	79.3
Python	146	262	116	79.5
Mixed	257	461	204	79.4

Table 3.3: Number of entries for 5 line restriction and 15 line restriction per language.

3.3.4 Evaluating the Effect of Dataset Size on Model Performance

To assess how dataset size affects machine learning model accuracy, we used the largest available dataset, Mixed 15 Lines, and created a range of datasets from 10% to 100% in increments of 10%. Each dataset was split into train, test, and validation sets, and the process was repeated five times to account for randomization. The model was trained on each dataset, and accuracy was measured using EM and BLEU scores. The mean of the five training results was plotted as a scatter plot with dataset size on the x-axis and accuracy on the y-axis for both EM and BLEU scores. To provide a better understanding of the variance in our results, we added error bars to the scatter plot representing the minimum and maximum accuracy achieved in each experiment.

The results can be observed in Figure 3.3. Both plots showed a positive correlation between dataset size and model accuracy. This experiment suggest that larger datasets could have a positive impact on performance.

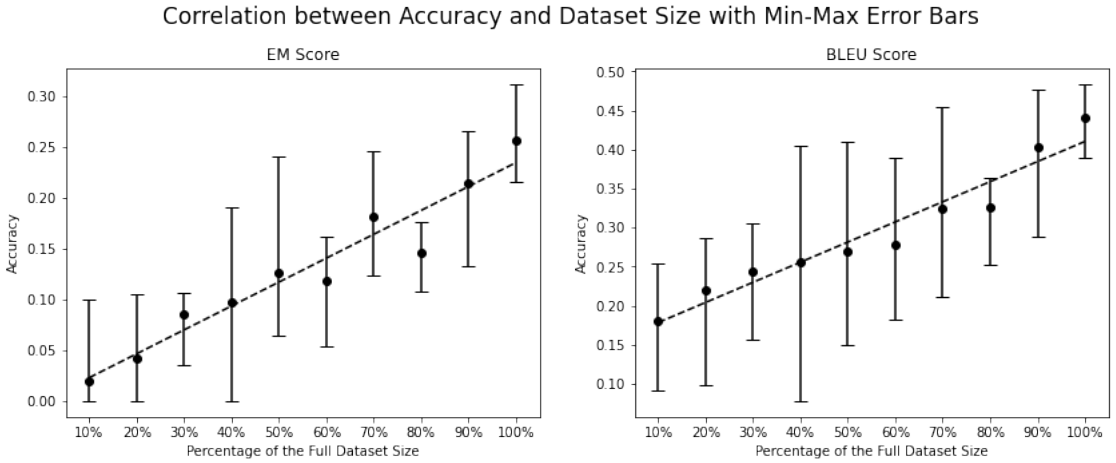

Figure 3.3: Scatterplot of accuracy vs dataset size with min-max error bars.

3.3.5 Evaluating the Effect of Lines Changed on Model Performance

After observing the impact of dataset size on model performance, we were interested in exploring whether relaxing the restriction on the number of changed lines would have a similar effect, and also temporarily introduced an Unrestriced dataset for exploration purposes. The number of entries for the unrestriced lines change is 348, 485 and 833 for Java, Python and Mixed respecively. We acknowledge that this is a complex problem, as we not only increase the dataset size, but also allows more lines to be changed, which can have certain effects on model performance.

The graphs in Figure 3.4 confirms our suspicion of the complexity. We can observe that 5 Lines perform better than 15 and Unrestricted on Random split despite being the smallest dataset. For the Time split, 15 Lines seems to perform the best. However, do take note of the y-axis, as the Time split's score is generally much lower than Random split. Drawing insights from this experiment, we can motivate why we decided to restrict the number of lines, as it appears to improve model performance. However, as we noted in the section on external validity, this could potentially affect the model's generalizability.

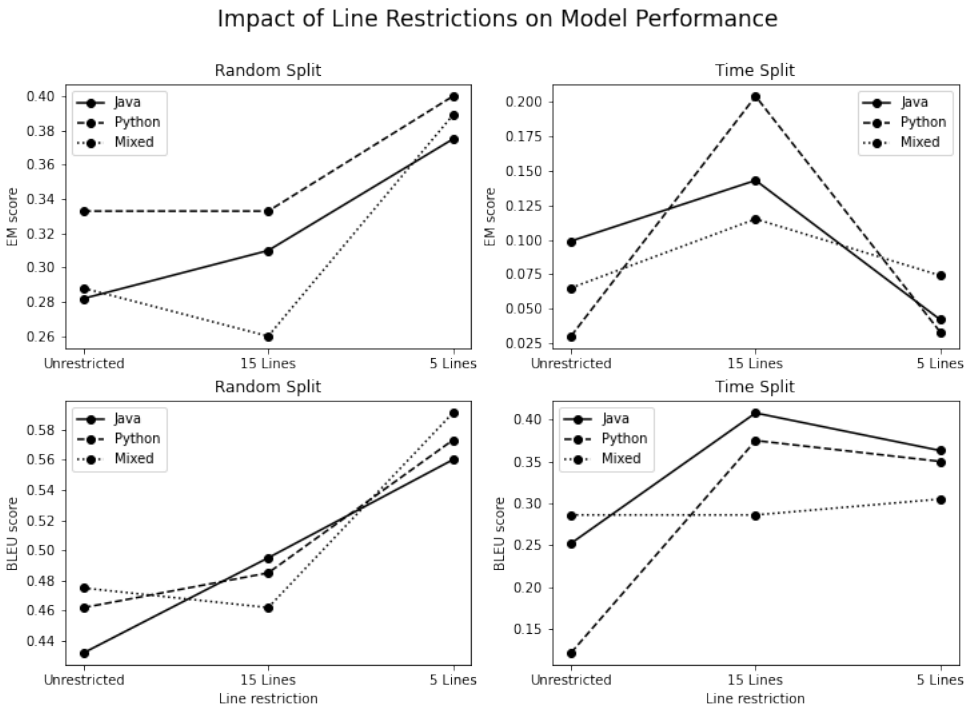

Figure 3.4: Scatter plots on performance based on lines changed.

Chapter 4
Results

In order to discuss our defined research questions, we will now present the results, which will be the basis for identifying the key factors that affect the model performance. The results will also be used to carry a discussion when comparing our work with related work. The insights that we have gained so far in our exploratory experiments, suggest that augmenting the dataset size may lead to improved model performance, but at the same time, it may introduce additional complexity. Bearing these insights in mind, we will analyze the results to investigate the various factors that affect model performance.

4.1 Model Performance

Table 4.2 presents the model performance, which is consistent with the results of the experiment described in Section 3.3.5, except for the fact that we did not use the Unrestricted dataset as it was solely for exploratory purposes. We have also provided our results visually in Figure 4.1 for readability. Datasets split on Random (diagonal and white bars) outperformed those split on Time (dots and black bars), and 5 Lines performed better than 15 Lines in the Random split, while the opposite was true for the Time split. Similar patterns were observed for the BLEU score, although it should be noted that this is only a supplementary metric, but could provide an indication of how close the prediction is to target.

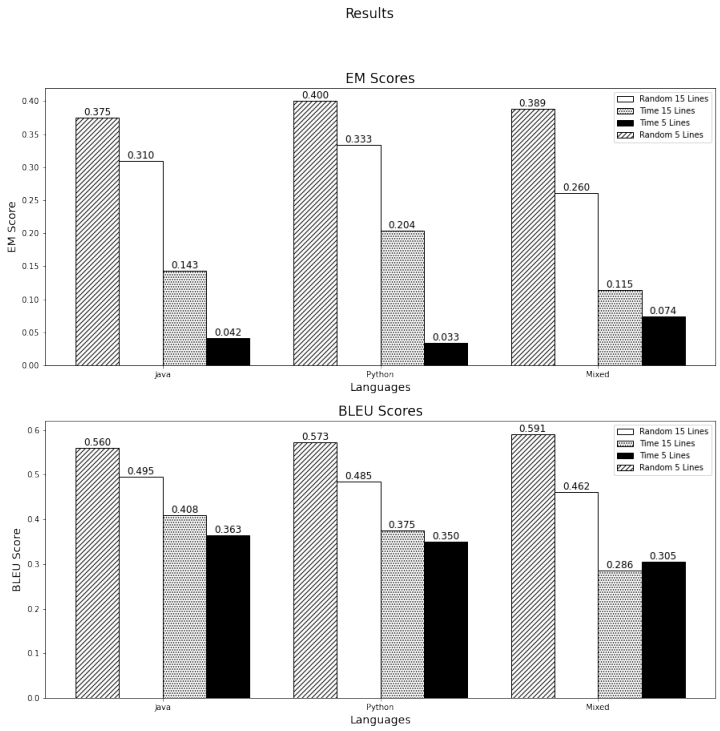

Figure 4.1: Model performance visualized as bar plots.

EM				
	Random		**Time**	
Language	**5 Lines**	**15 Lines**	**5 Lines**	**15 Lines**
Python	**0.400**	0.333	0.033	0.204
Java	**0.375**	0.310	0.042	0.143
Mixed	**0.389**	0.260	0.074	0.115

(a) EM Scores for Python, Java, and Mixed on Random and Time split with Line Change Restrictions of 5 and 15. The best scores are highlighted in bold.

BLEU				
	Random		**Time**	
Language	**5 Lines**	**15 Lines**	**5 Lines**	**15 Lines**
Python	**0.573**	0.485	0.350	0.375
Java	**0.560**	0.495	0.363	0.408
Mixed	**0.591**	0.462	0.305	0.286

(b) BLEU Scores for Python, Java, and Mixed on Random and Time split with Line Change Restrictions of 5 and 15. The best scores are highlighted in bold.

Figure 4.2: Model Performance Results

4.1.1 Examples of Raw Predictions

In this section, we will provide some examples of what the model can generate, showcasing both correct and incorrect predictions. Additional examples are also included in Appendix B. In Section 3.1.3, we presented an example of a vulnerability and its corresponding fix. While the generated code in the example below is incorrect, it provides insight into the typical code generated by the model for incorrect predictions. The example shown in Listing 4.1 highlights a vulnerability related to CWE-287 (Improper Authentication), where the vulnerable code compared a plain password with a user password and returned True if they matched, which is generally considered unsafe. The target code replaced the vulnerable comparison with 'secrets.compare_digest', a more secure method to compare two strings. However, the raw prediction for the 5 Lines dataset appeared to be a function used to parse JSON data from a remote source. The raw prediction for the 15 Lines dataset checked whether the user password started with a specific string. Both predictions are unrelated to password comparison, but looks like it could be correct at first glance.

```
# Vulnerable code
CWE-287
def verify_password(plain_password, user_password):
    if <S2SV_StartBug> plain_password == user_password: <S2SV_EndBug>
        LOG.debug("password true")
        return True
    return False

# Target
<S2SV_ModStart>
secrets.compare_digest(plain_password,user_password):
<S2SV_ModEnd>

# Incorrect raw prediction 5 Lines
<S2SV_ModStart> defusedxml.ElementTree.from_remote_json(user_password))
<S2SV_ModEnd>

# Incorrect raw prediction 15 Lines
<S2SV_ModStart>
if user_password and not user_password.startswith("http://pypi.python.
    org"):
    LOG.debug("password true")
    return _is_valid_hostname(plain_password)
else:
LOG.debug("password false")
<S2SV_ModEnd>
```

Listing 4.1: Example of incorrect raw predictions made by Python-5-Time and Python-15-Time.

Let us take a closer look at another example of a vulnerability and its corresponding fix. In this

case, the vulnerability is related to CWE-74 (Injection), and the function '_get_unauth_response' is used to generate a response when an unauthenticated user attempts to access a protected page. Listing 4.2 shows that the 15 Line model generated a correct prediction by modifying the code to escape the 'reason' parameter, which prevents an attacker from injecting malicious scripts into the response. However, the 5 Line model's prediction was incorrect and did not address the vulnerability, but again looks like real code.

```
# Vulnerable code
CWE-74
def _get_unauth_response(self, request, reason):
    if request.is_ajax():
        return HttpResponseForbidden(json.dumps({"error": force_text(reason)}))

    error_params = urlencode({"error": force_text(reason)})
    login_url = force_str(reverse("shuup_admin:login") + "?" + error_params)
    resp = redirect_to_login(next=request.path, login_url=login_url)

    if is_authenticated(request.user):
        raise Problem(_("Can't view this page. %(reason)s") % {"reason": <S2SV_StartBug> reason}).with_link( <S2SV_EndBug>
            url=resp.url,
            title=_("Log in with different credentials...")
        )

    return resp

# Target and correct raw prediction for 15 Lines
<S2SV_ModStart> escape(reason)}).with_link( <S2SV_ModEnd>

# Incorrect raw prediction 5 Lines
<S2SV_ModStart>
safe_redirect(request, "safe_redirect", login_url=login_url,
method=\'POST\')
<S2SV_ModEnd>
```

Listing 4.2: Example of correct and incorrect raw predictions made by Python-5-Random and Python-15-Random

These examples will hopefully shed some light on the models we have developed for this project. To gain further insights into the various factors that impact model performance, we will now present details about the dataset distributions.

4.2 Distribution of Repositories

The results in Section 4.1, indicate that the choice of split method significantly affected model performance, prompting us to investigate further. Upon examining the repository distributions, we discovered that the Random split had more repository overlap than Time split. We visualized this phenomenon in Figures 4.3 to 4.6. For ease of comprehension, we grouped repositories with the same distribution and provided additional details in Appendix D. Although we omitted the graphs for the Mixed dataset, we included Table C.1 in Appendix C for reference purposes.

The most notable observation from the plots is the difference between the top bar (Time split) and the bottom bar (Random split). The top bar consistently showed less overlap between the train (white), validation (black), and test (dots) sets, while the Random split had more overlap across all models. This suggests that random splitting increases the likelihood of methods from the same repository appearing in different splits. If these methods are similar within their repository, this could inflate model performance. The plots are also annotated with number of correct predictions, to show that most of the correct predictions come from the bigger repositories.

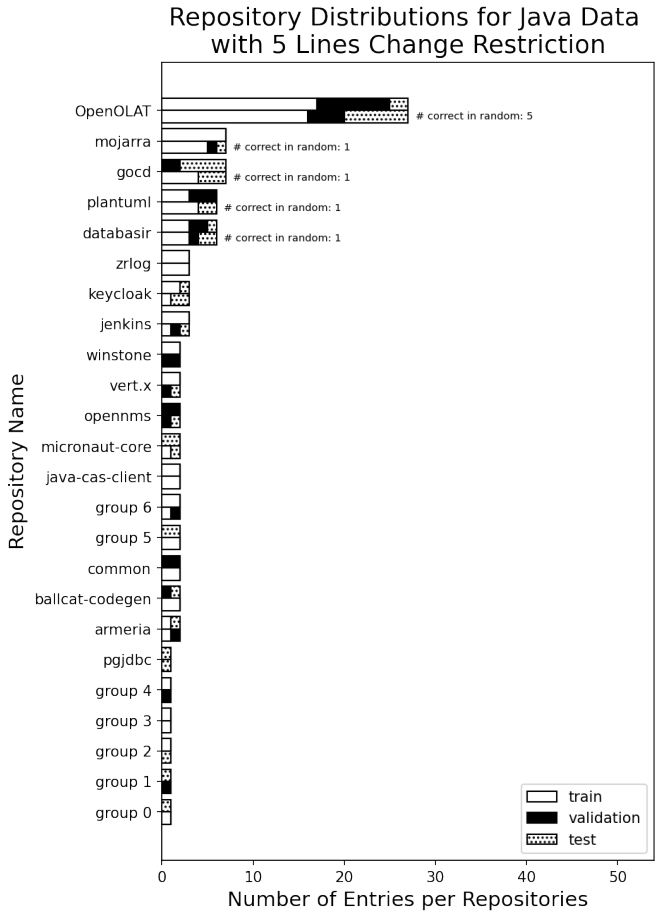

Figure 4.3: Repository distribution for Java 5 Lines. The top bar represents the Time split and the bottom bar is the Random split. Details about the groups in Appendix D.

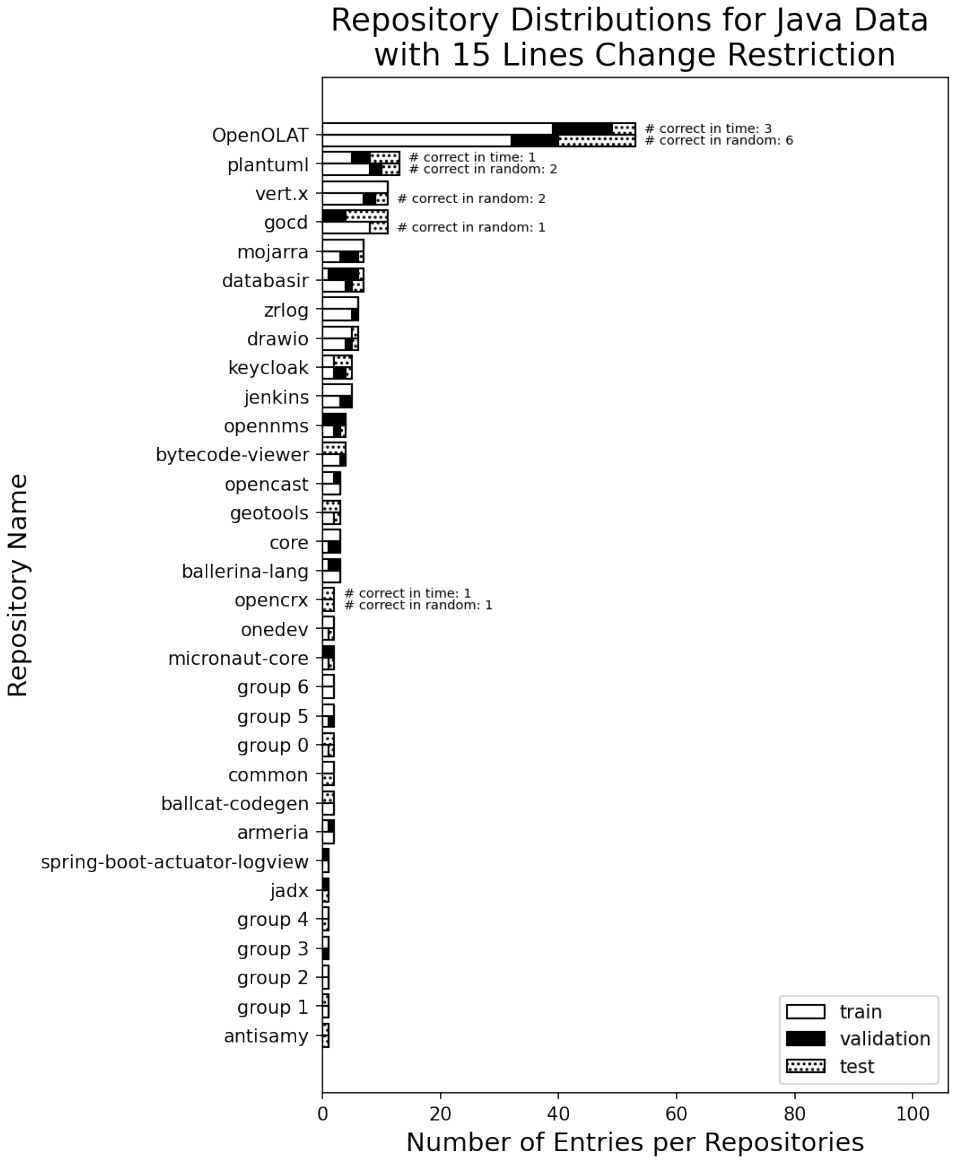

Figure 4.4: Repository distribution for Java 15 Lines. The top bar represents the Time split and the bottom bar is the Random split. Details about the groups in Appendix D.

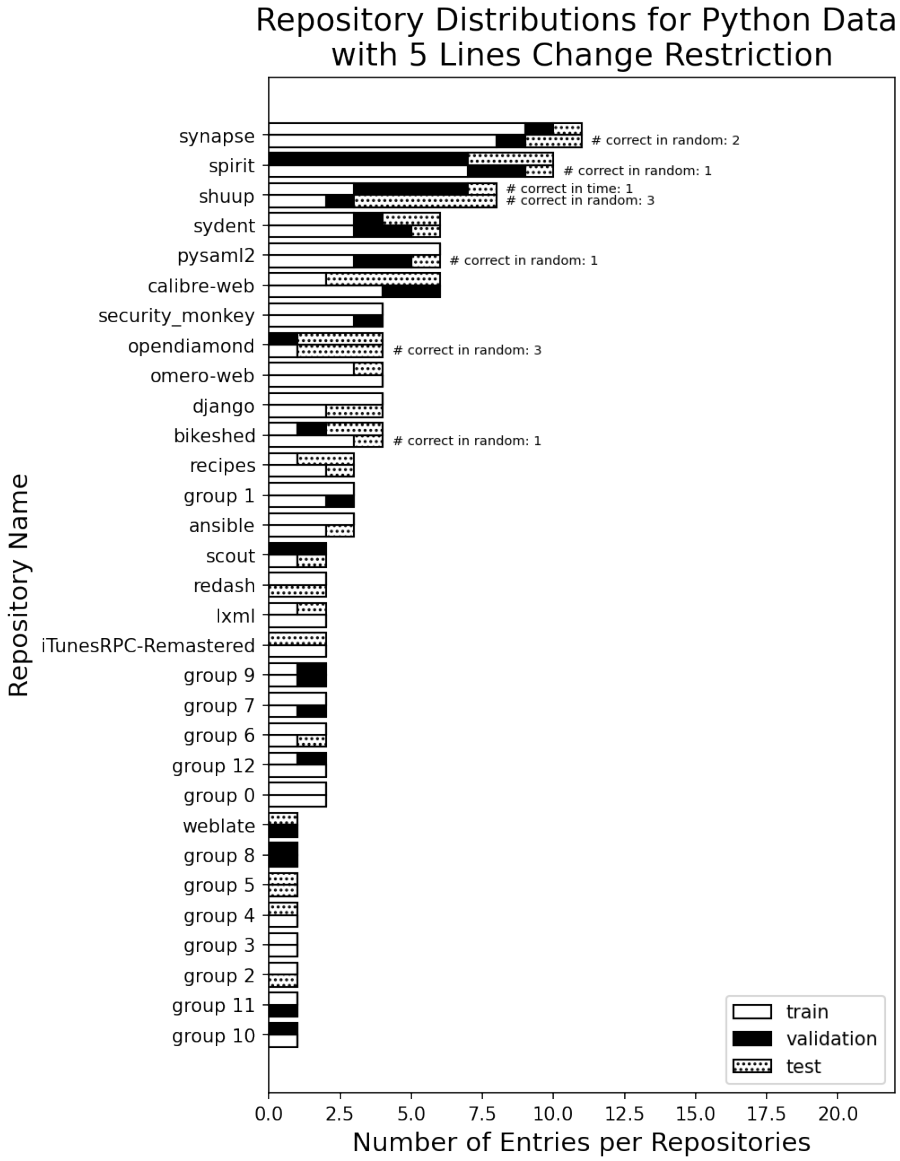

Figure 4.5: Repository distribution for Python 5 Lines. The top bar represents the Time split and the bottom bar is the Random split. Details about the groups in Appendix D.

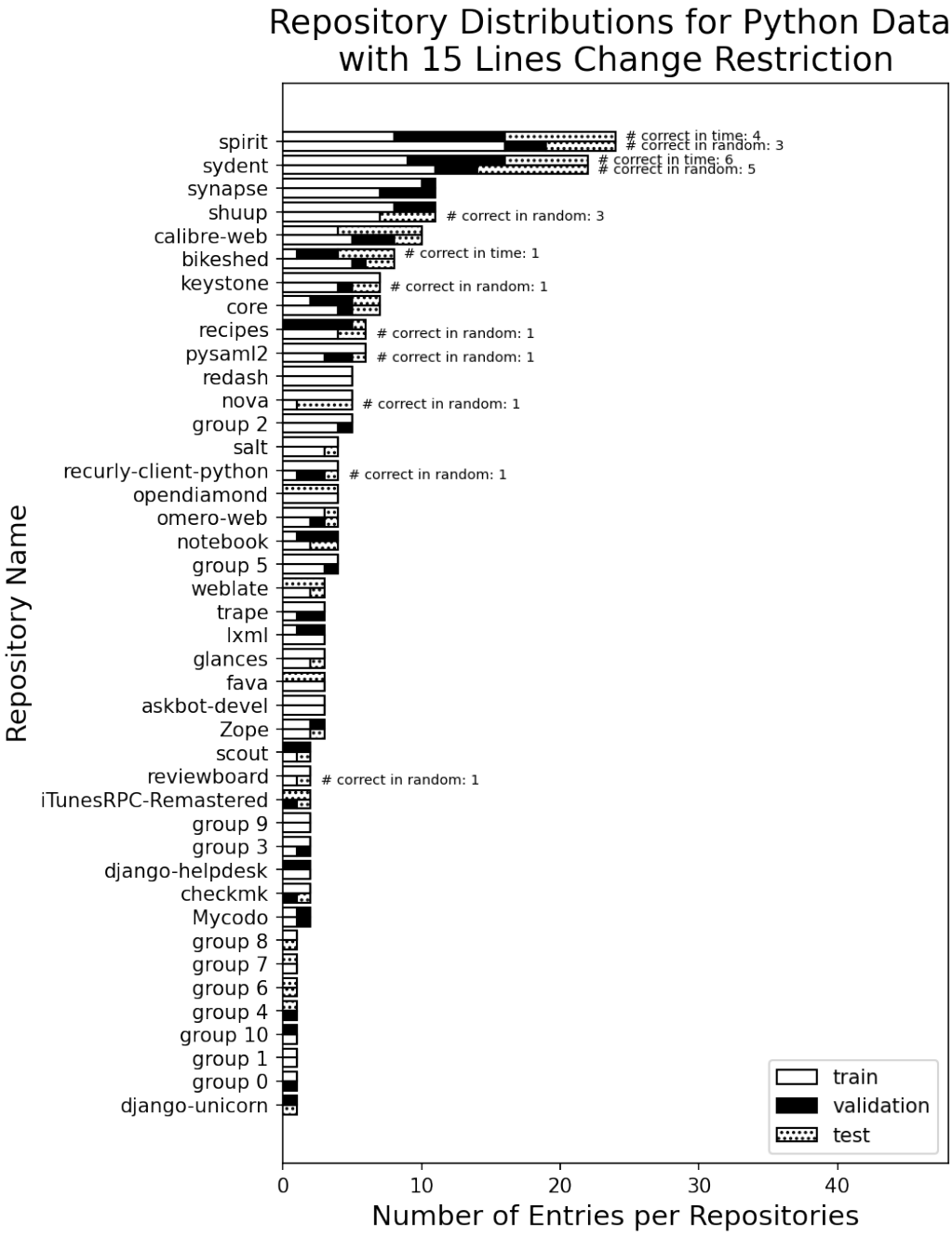

Figure 4.6: Repository distribution for Python 15 Lines. The top bar represents the Time split and the bottom bar is the Random split. Details about the groups in Appendix D.

4.3 Distribution of Commit Years

To show what the years of the different split methods look like, we have created Figures 4.7 and 4.8. These figures show that for Random split, the commit years are spread out across train, validation and test. For the Time split, the training data mostly consist of earlier entries, whereas the validation and test split contains newer data. These insights could explain

why there could be a bigger chance of overlap in repositories when we split Randomly instead of Time, as we highlighted in the previous section. Note that this statement is only valid if we assume that the commit years within the repositories are not spread out. Also note that the splits are not strictly chronological as we only group by year. As can also be observed, there was an increase in number of entries in the dataset from before 2020 and after 2020, which means that we have most data from later years, which can affect our generalizability.

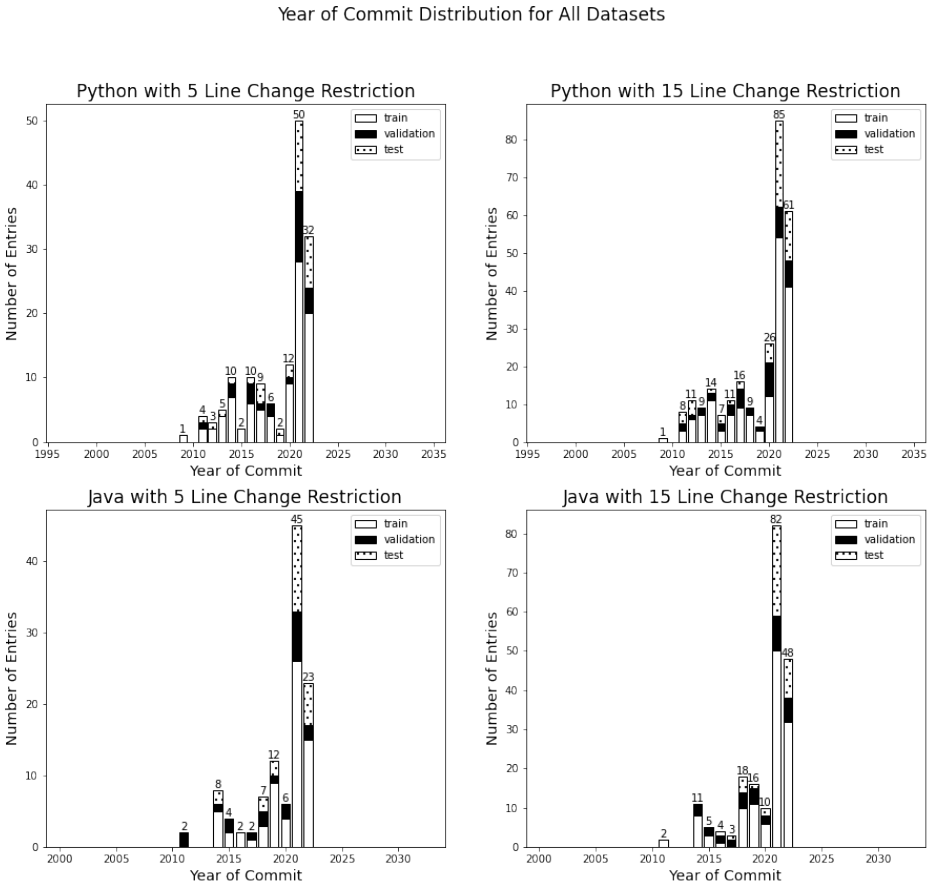

Figure 4.7: Distribution of commit years across all datasets in the Random split data.

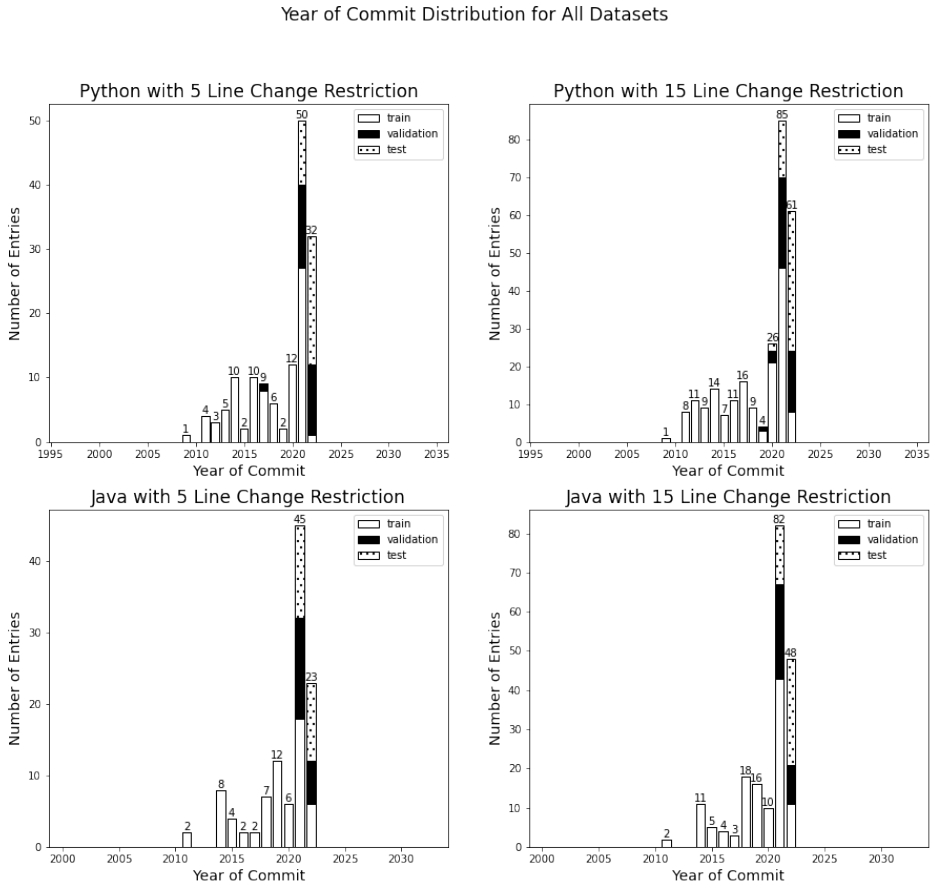

Figure 4.8: Distribution of commit years across all datasets in the time split data.

4.4 Connection CWE Categories and Repositories

After observing the potential influence of split methods on repository distribution and the dominance of larger repositories in producing correct predictions, we investigated the potential connection between CWE categories and repositories. Our findings are summarized in Table 4.1, which reveals a trend of certain repositories having a high number of correct predictions for each CWE category. For instance, the dominant repository for CWE-74 is **OpenOLAT**, the biggest repository for Java as shown in Figures 4.3 and 4.4. Similarly, **spirit** is the most frequent for CWE-610, consistent with the Python data in Figures 4.5 and 4.6. For CWE-20, **sydent** is the leading repository and one of the largest repositories. CWE-706 and CWE-298 are mainly consising entries from **opendiamond** and **keystone** respectively, both of which are relatively large repositories. This suggests that many entries in these repositories may belong to the same category.

CWE Category	Java				Python				Mixed			
	Random		Time		Random		Time		Random		Time	
	5 Lines	15 Lines	5 Lines	15 Lines	5 Lines	15 Lines	5 Lines	15 Lines	5 Lines	15 Lines	5 Lines	15 Lines
CWE-74	**OpenOLAT 3** gocd 1	**OpenOLAT 5** gocd 1 opencrx 1	opencrx 1	**OpenOLAT 3** opencrx 1	shuup 3 reviewboard 1 synapse 1	shuup 3 reviewboard 1	shuup 1	bikeshed 1	shuup 3 **OpenOLAT 3** gocd 2 synapse 1 reviewboard 1	**OpenOLAT 5** shuup 2 reviewboard 1 gocd 1 opencrx 1	weblate 1 shuup 1 opencrx 1	**OpenOLAT 2** omero-web 1 opencrx 1
CWE-287						**keystone 1**				**keystone 1**		
CWE-610	plantuml 1	plantuml 2	plantuml 1		pysaml2 1 synapse 1 **spirit 1**	**spirit 3** sydent 3 recipes 1 recurly- client- python 1 pysaml2 1		**spirit 4**	**spirit 1** synapse 1 plantuml 1 pysaml2 1	**spirit 3** plantuml 2 sydent 1 recurly- client- python 1 pysaml2 1		**spirit 4**
CWE-706	OpenOLAT 2 mojarra 1	xwiki- platform 1 OpenOLAT 1		xwiki- platform 1	**opendiamond 3** bikeshed 1	ganga 1			**opendiamond 3** OpenOLAT 2 bikeshed 1 mojarra 1	OpenOLAT 1 xwiki- platform 1 ganga 1		xwiki- platform 1 **opendiamond 1**
CWE-20	databasir 1	vert.x 2				**sydent 2** nova 1		**sydent 6**	databasir 1	vert.x 2 **sydent 1**		**sydent 1**

Table 4.1: Overview of which repositories the correct predictions come from in each CWE category. The leading repositories are bolded.

4.5 Distribution of CWE-categories

After observing the impact of the split method on repository distribution and its connection to CWE categories, we now aim to investigate the patterns within individual CWE-IDs of the categories. We found that correct predictions mostly come from the biggest CWE-categories, which is not surprising. For Java, the biggest category is CWE-74 (Improper Neutralization of Special Elements in Output Used by a Downstream Component), while for Python, it is CWE-610 (Externally Controlled Reference to a Resource in Another Sphere). Note that this is consistent with the results from the previous section, as the correct predictions in these categories mostly correspond to the biggest repositories, **OpenOLAT** for Java and **spirit** for Python.

To simplify the information and visualize the distribution of individual CWE-IDs within each category, we present Figures 4.9 and 4.10, which depict Tables 4.2 and 4.3. The bars are also annotated with the number of correct predictions, to highlight that it is the bigger CWE-IDs that usually have correct predictions. The introduction of 15 Lines reveals some new CWE-IDs with a relatively lower number of entries, which could affect the model performance as the CWE-IDs become more diverse.

Java								
Category	**CWE-ID**	**Number of correct predictions**				**Number of Entries**		**Increase**
		5 Lines		**15 Lines**		**5 Lines**	**15 Lines**	
		Random	**Time**	**Random**	**Time**			
CWE-74	CWE-79	1	1	2	1	17	28	11
	CWE-91	3	0	5	3	14	29	15
	CWE-74	0	0	0	0	11	12	1
	CWE-77	0	0	0	0	4	5	1
	CWE-89	0	0	0	0	3	5	2
	Total	**4**	**1**	**7**	**4**	**49**	**79**	**30**
CWE-706	CWE-22	3	0	2	1	26	52	26
	Total	**3**	**0**	**2**	**1**	**26**	**52**	**26**
CWE-287	CWE-287	0	0	0	0	3	8	5
	CWE-798	0	0	0	0	2	2	0
	CWE-290	0	0	0	0	2	2	0
	CWE-306	0	0	0	0	1	3	2
	Total	**0**	**0**	**0**	**0**	**8**	**15**	**7**
CWE-20	CWE-20	1	0	2	0	14	27	13
	Total	**1**	**0**	**2**	**0**	**14**	**27**	**13**
CWE-610	CWE-611	0	0	0	0	6	7	1
	CWE-918	1	0	2	1	8	19	11
	Total	**1**	**0**	**2**	**1**	**14**	**26**	**12**

Table 4.2: CWE-distribution for Java: 5 Lines Restriction and 15 Lines Restriction

Python								
Category	**CWE-ID**	**Number of correct predictions**				**Number of Entries**		**Increase**
		5 Lines		**15 Lines**		**5 Lines**	**15 Lines**	
		Random	**Time**	**Random**	**Time**			
CWE-74	CWE-79	4	1	4	0	33	55	22
	CWE-88	0	0	0	0	1	1	0
	CWE-74	1	0	0	0	4	3	-1
	CWE-77	0	0	0	0	1	2	1
	CWE-89	0	0	0	0	3	5	2
	CWE-1236	0	0	0	0	0	3	3
	CWE-78	0	0	0	1	3	8	5
	Total	**5**	**1**	**4**	**1**	**45**	**77**	**32**
CWE-706	CWE-22	4	0	1	0	19	29	10
	CWE-59	0	0	0	0	4	8	4
	Total	**4**	**0**	**1**	**0**	**23**	**37**	**14**
CWE-287	CWE-287	0	0	1	0	3	17	14
	CWE-295	0	0	0	0	1	1	0
	CWE-522	0	0	0	0	1	1	0
	Total	**0**	**0**	**1**	**0**	**5**	**19**	**14**
CWE-20	CWE-20	0	0	3	6	19	41	22
	Total	**0**	**0**	**3**	**6**	**19**	**41**	**22**
CWE-610	CWE-611	1	0	1	0	9	10	1
	CWE-918	0	0	5	0	13	31	18
	CWE-601	2	0	3	4	32	47	15
	Total	**3**	**0**	**9**	**4**	**54**	**88**	**34**

Table 4.3: CWE-distribution for Python: 5 Lines Restriction and 15 Lines Restriction

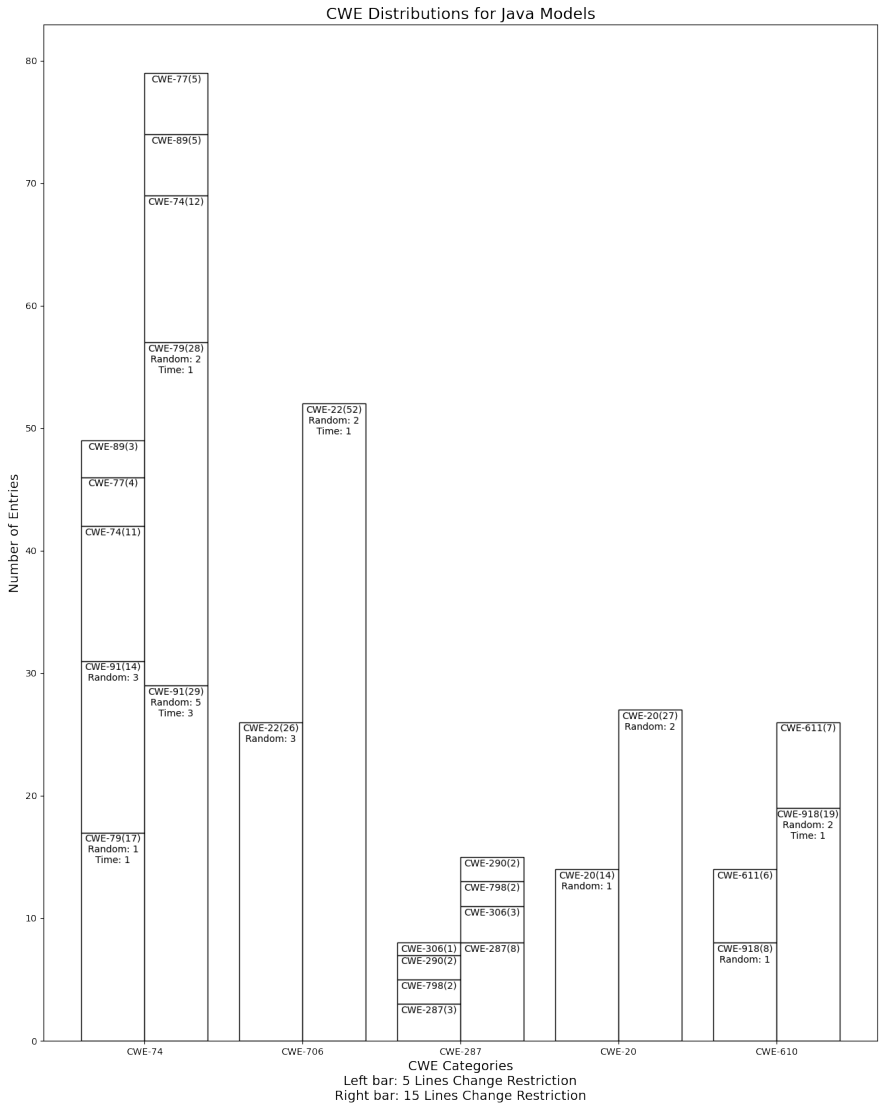

Figure 4.9: This graph shows the number of entries per CWE-category for Java. The height of each bar corresponds to the number of occurrences of the CWE categories, with the count of each CWE-ID noted inside parentheses. Additionally, the chart displays the number of correct predictions for the Random and Time datasets.

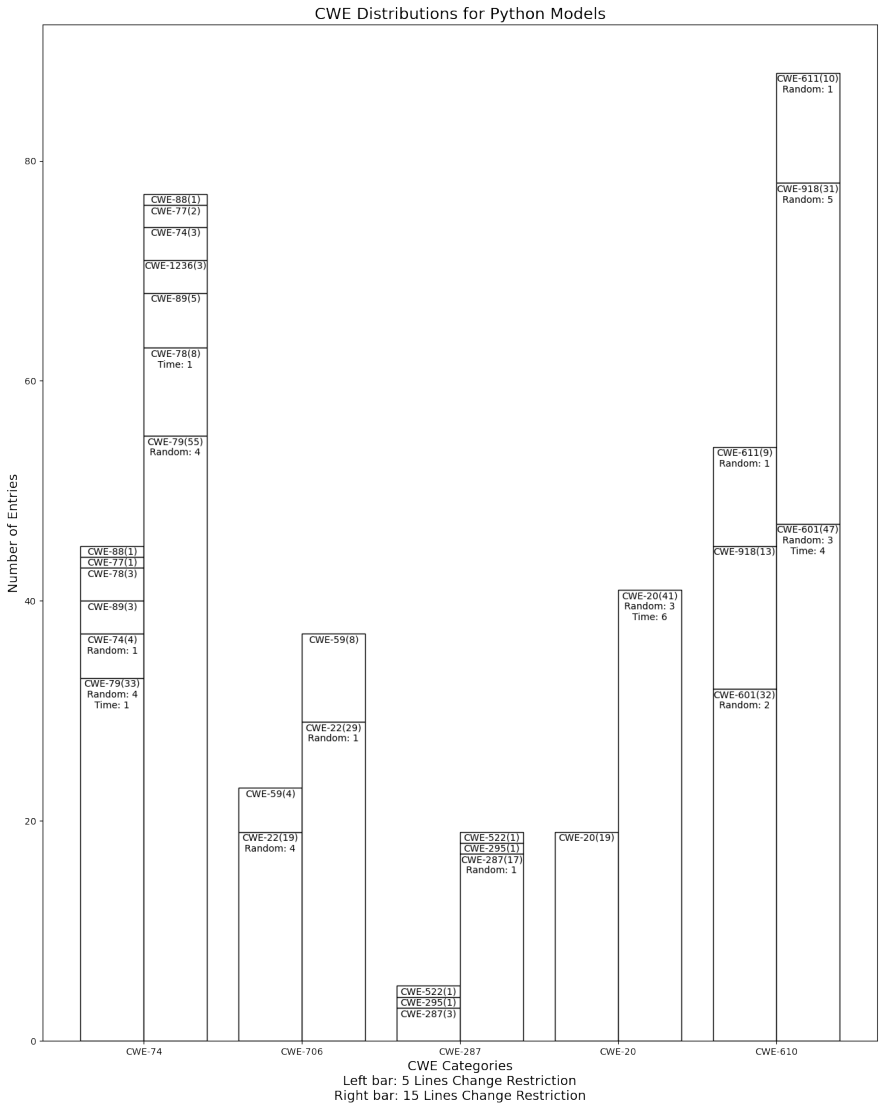

Figure 4.10: This graph shows the number of entries per CWE-category for Python. The height of each bar corresponds to the number of occurrences of the CWE categories, with the count of each CWE-ID noted inside parentheses. Additionally, the chart displays the number of correct predictions for the Random and Time datasets.

4.6 Distribution of Lines Changed

Last but not least, we wanted to look into the lines changed in the dataset. Specifically how many lines added and deleted the correct predictions had. Figures 4.11 to 4.13 illustrate the distribution of line additions and deletions, at method level, across all datasets (bottom graph) and in correct predictions (top graphs). To make the diagrams easier to read, we have categorized the patches for 5 Lines as short with 0-2 lines added or deleted and 3-5 lines as long. For 15 Lines, short is considered 0-7 lines and 8-15 lines for long. To see the ungrouped plots we refer to Appendix E.

Across all datasets, the majority of lines added or deleted for 5 Lines were 0 to 2 lines of code. With 15 Lines, there were mainly 0 to 7 lines changed. Notably, these graphs do not indicate any relationship between line additions and deletions. The distribution of correct predictions (top graphs) roughly follows the shape of the overall datasets (bottom graphs). These graphs indicate that we have a bias towards shorter patches in general in the dataset.

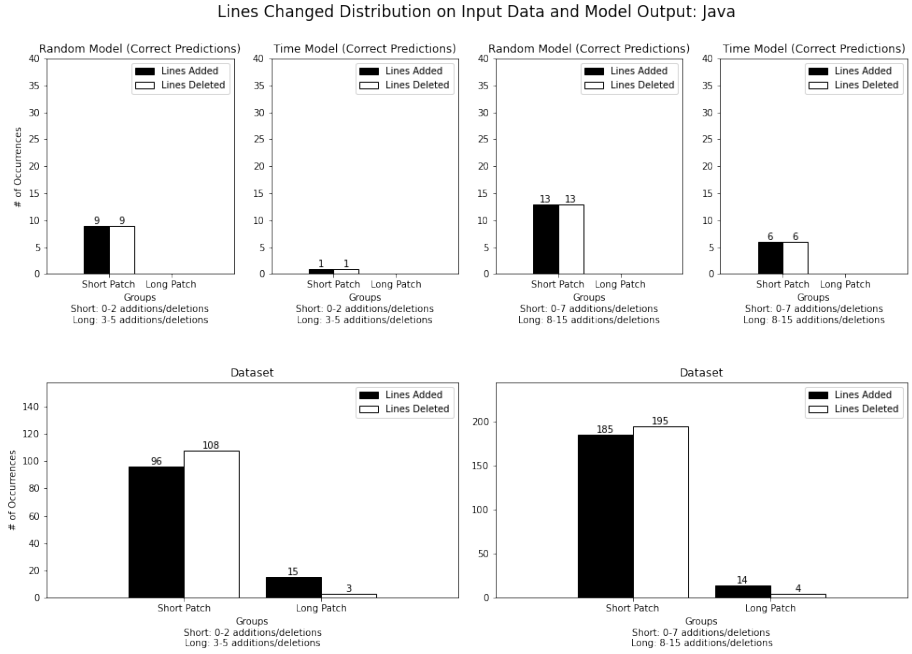

Figure 4.11: Comparison of line additions and deletions in correct predictions for Java 5 Line (left plots) and Java 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

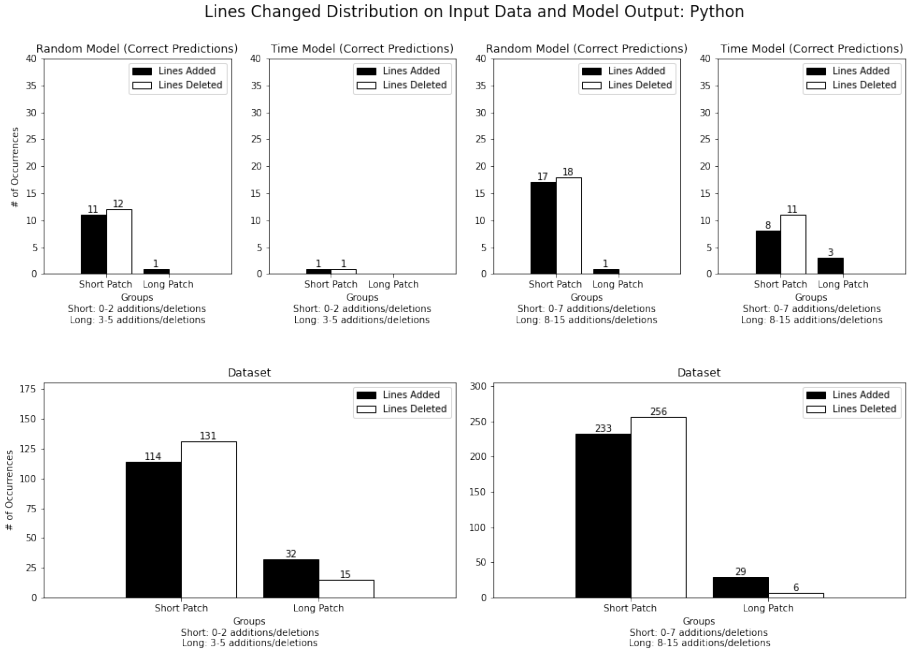

Figure 4.12: Comparison of line additions and deletions in correct predictions for Python 5 Line (left plots) and Python 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

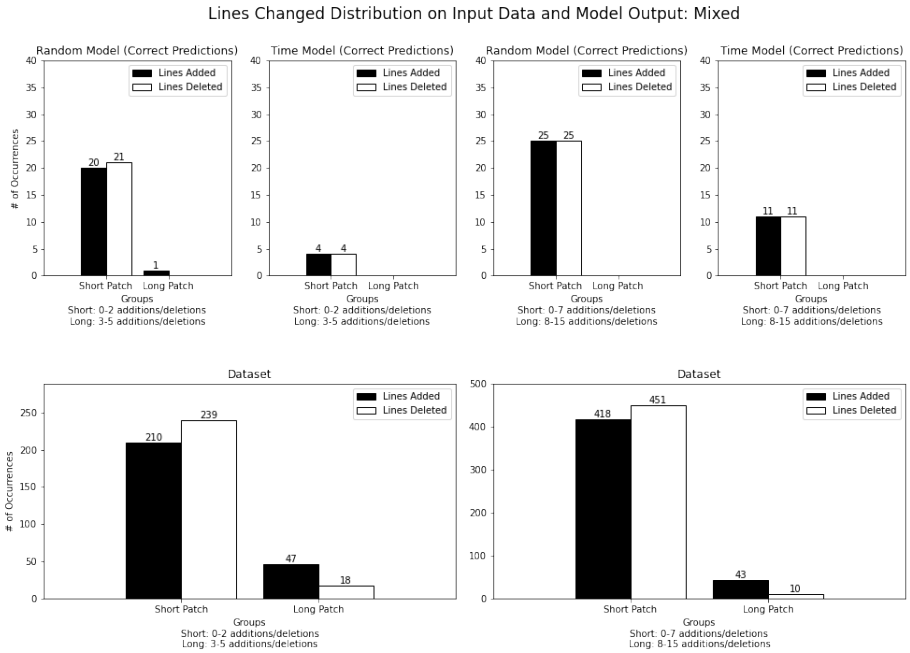

Figure 4.13: Comparison of line additions and deletions in correct predictions for Mixed 5 Line (left plots) and Mixed 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

Chapter 5
Discussion

Having presented all the relevant results, we can now interpret the different factors that we have introduced and examine how they affect model performance. As a reminder, the factors we have discussed previously include dataset size, split method (Random vs Time), repository- and CWE-distributions, and the number of lines changed in the correct predictions. With these factors in mind, we aim to address our first research question, which focuses on identifying the key factors that influence model performance. Finally, we will conclude this chapter by comparing our results to the related work, which was the second research question we aimed to explore for additional insights.

5.1 Influence of Dataset Size

Initially, we invested considerable effort into augmenting our dataset size to improve our results, based on our belief that a larger dataset would enhance model performance (Section 3.3.4). To this end, we adopted two approaches: 1) merging the Java and Python datasets to form a Mixed dataset, and 2) increasing the number of changed lines from 5 to 15. However, we soon realized that merely increasing the dataset size did not result in improved performance, as can be seen in Figure 4.1 suggesting that the underlying issue was more intricate than we initially assumed.

Regarding the Mixed dataset, while the results were not significantly improved, they were also not notably worse than those obtained for the individual languages. This implies that even though we introduced more variables in the Mixed dataset, the performance remained acceptable. Our observations suggest that this approach may be effective, as the model might have learned patterns from different languages and been exposed to more diverse repositories, which could enhance its generalizability. This could also indicate that the individual languages results are potentially inflated and possibly overfitted.

Regarding the number of lines changed, we observed mixed results depending on the split method used. For the Random split, the smaller 5 line dataset performed better, while

for the Time split, the 15 line dataset yielded better results. We can only speculate that the 5 line changes might be "easier" for the model to generate since the added or deleted lines were mostly 0-2 lines, although it is not necessarily true that shorter patches are easier to fix. In contrast, the 15 line changes had 0-7 lines changed per patch, suggesting that they are typically longer patches. However, assuming that fixes are easier for shorter patches is just one interpretation, and it could also be due to having more entries within the shorter patches, resulting in better performance. Furthermore, we are uncertain as to why the 15 lines dataset performed better than the 5 lines dataset for the Time split. Further investigation is necessary to draw a definitive conclusion.

5.2 Influence of Split Method

The split method had a significant impact on our model, and we want to provide more details about it. As we observed, the Random split generally performed better than the Time split. However, it's important to note that the time factor is crucial for our study, which relied on open-source data. Thus, our discussion on the relevance of the Time split applies only to this specific context. It would be interesting to investigate the effects of the Time split on proprietary data, but unfortunately, we didn't have access to such data.

5.2.1 Time dependency

The Time split models consistently performed worse than the Random split models. This phenomenon might be interesting to research further on, as it could be an indication that our dataset may be subject to time-dependency, which means that using older security patches as predictors of future vulnerabilities may not be as effective. If this is the case, we would need to keep the time aspect in mind while designing the data to develop a model that can generalize well to unseen data.

As noted in the Background section, vulnerabilities in open-source software tend to be discovered and reported sooner than those in proprietary software. Open-source vendors also reportedly release security vulnerability patches sooner than closed-source vendors. With these insights, we find it reasonable to assume that vulnerabilities in OSS may evolve more rapidly compared to proprietary software. This is because the open-source community might have faster access to vulnerabilities, hence allowing attackers to adopt new ways of exploiting vulnerabilities.

If this hypothesis regarding time dependency is accurate, it would likely only apply to open-source software. Based on the differences in their patching behaviors and the transparency of the open-source community, we can speculate that vulnerabilities in open-source software may evolve at a faster rate than those in proprietary software. However, to confirm this hypothesis, further research is required.

5.2.2 Influence on Repositories

We also want to discuss how the split method affected the distribution of commit years in the splits and ultimately the repositories. As expected, the Random and Time splits follow different year distributions, with the training data consisting of older data and the validation

and test data consisting of newer data in the Time split. In contrast, the years were uniformly distributed in the Random split. It is worth noting that we have the most data on years from 2020 and onwards when we examine the year distributions. This is important because it could potentially bias our model to generate well for vulnerabilities that were committed during those years.

Interestingly, the year split seems to have a significant impact on the repositories. We suspect that the repositories themselves contain entries from the same year, making it harder for the Time split to train on data from the same repository than for the Random split. This could be both a good and a bad thing, as it is unclear whether the improved performance of the Random split can be attributed to overfitting, where the model has seen many examples from the same repository.

The main finding is that the Random split has more overlap in entries from different repositories across the train, validation, and test sets than the Time split. This is particularly relevant when the repositories themselves are large and contain many entries. To reduce bias towards certain repositories, it is important to strive for less overlap in the splits. By doing so, we can increase the model's ability to learn general patterns in vulnerabilities, which can improve its overall performance.

5.3 Influence of CWE-distribution

We have discussed how various factors such as dataset size, split method, repository distribution, and number of lines changed have impacted our model's performance. Lastly, we would like to briefly touch upon the distribution of Common Weakness Enumerations (CWEs) in our dataset.

As we discovered in the Results section, the number of correct predictions mainly come from the biggest CWE categories (CWE-74 and CWE-610), and there also appears to be a connection between CWE-distribution and repository for the correct predictions. This may suggest that there are many methods within a repository that fix the same CWE category, or that certain repositories are more prone to weaknesses within specific CWE categories. However, we cannot definitively state what these correlations are or if they exist. Nevertheless, we observed a pattern that could potentially be interesting to investigate further.

We limited the number of CWE categories in our study to create a more focused model. This decision was based on the findings from Huang et al.'s study [8] and our own early experiments (see Section 3.1.1), which showed a positive impact on performance when limiting the number of CWEs. However, this decision resulted in some data loss of approximately 30% for the 5 Lines and 40% for the 15 Lines datasets. Despite this trade-off, limiting the CWE categories also made it easier for us to analyze the data and potentially detect patterns. But we want to highlight that limiting the CWE inevitably limits the model's generalizability.

Our findings suggest that the question of limiting the CWEs should be taken into account while ensuring that there are enough entries in the CWE-IDs, as very few entries do not seem to improve accuracy. Furthermore, since there could be a correlation between repository and CWE category, there is another argument for why it would be important to balance out the number of entries from each repository, to minimize bias. Another interesting approach is to balance the CWE-IDs uniformly and investigate if there are any differences in generating patches for different types of CWE-IDs, as we are curious if some categories may be "harder"

to predict than others.

5.4 Overview of the Impacting Factors on Accuracy

Before we proceed to address research question two, where we compare our results to related work, let us summarize our findings in regards to research question one. To facilitate the understanding of the different factors that might impact accuracy, we have created our version of a causal graph in Figure 5.1. In the diagram, the factors mentioned in this study are illustrated. When these factors are increased, the arrow with a plus sign (+) indicates an increase on model performance, while a minus sign (-) indicates a decrease on model performance. These factors can affect the model accuracy directly, by a direct arrow, or indirectly (e.g. split method indirectly affects model performance). It is important to keep in mind that a positive influence can result in inflated accuracy, which means that the model may perform well on the test set but be of limited practical use. And in contrast a negative influence could actually help improve the model's generalizability. As a result, we recommend balancing the dataset for the task intended for the model. Note that this causal graph represents our own interpretations on the impacting factors and could be subject for errors.

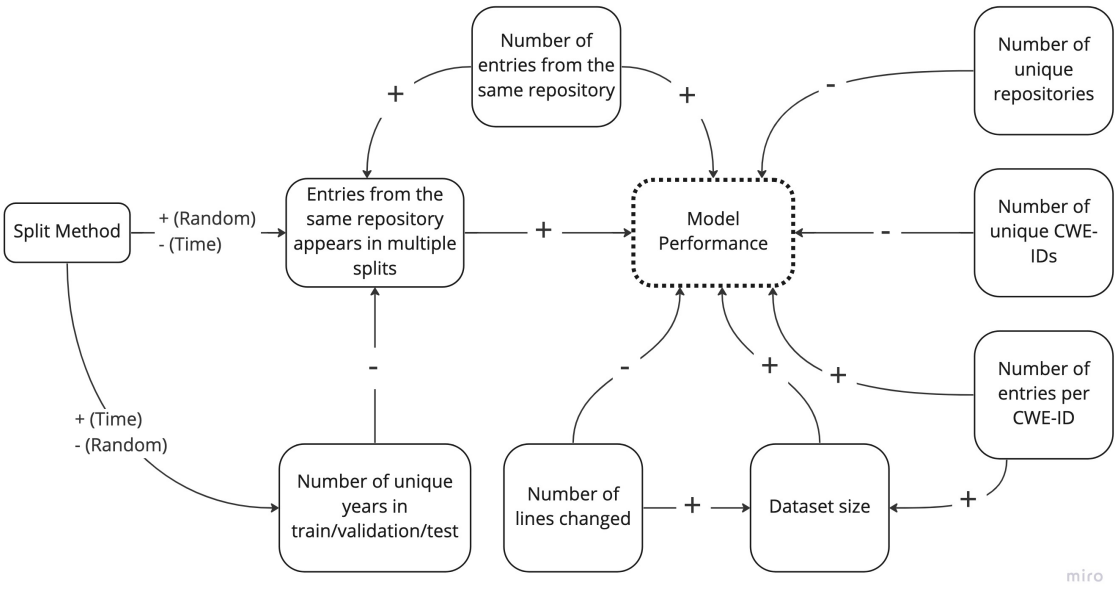

Figure 5.1: Causal graph of the influencing factors.

For illustration purposes, increasing the number of entries from the same repositories and repository overlap in the splits can result in inflated accuracy. Increasing the number of unique CWE-IDs and the number of lines changed in each function can influence performance negatively. These should only be decreased if the objective is to design a focused model that potentially achieves higher accuracy. Doing so requires caution, as this may come at the expense of the model's ability to handle a varied set of CWE-IDs and larger fixes, i.e. negatively impact its generalizability.

Overall, our study has made us aware of the importance of considering multiple factors that can affect model performance. This can hopefully contribute to some insights in how these factors can be optimized for the data engineering process, in order to develop a desired model in future research.

5.5 Comparison to Previous Work

Before we conclude this thesis, we would like to share our insights in comparing the results of our study to the related work of VulRepair [7] and Huang et al.'s study [8], which were introduced in Chapter 1.2.

5.5.1 Comparison to VulRepair

The Exact Match score of VulRepair is reported to be 44%, which we will compare to our own results. Our best performing model for 5 Lines was Python, which achieved a score of 40%, followed closely by Mixed (38.9%) and Java (37.5%). For 15 Lines, the Python model remained the best performer, achieving a score of 33.3%, followed by Java (31%) and Mixed (26%).

The results for Random 5 Lines are slightly lower than those of VulRepair but still in the comparable range. However, for Random 15 Lines, we saw a bigger difference in performance compared to VulRepair. There are a few improvement potentials for our study, as we only split the data in one go and did not train the model on different samples. This makes our results heavily dependent on the split that we got. To mitigate this, we could perform more samplings and present an average performance, which would be a more reliable result. Furthermore, due to resource limitations, we were only able to train with 30 beams while VulRepair had 50, something that could potentially positively affect their performance.

However, our datasets were significantly smaller than VulRepair's, which had 8,482 samples (nearly 18 times larger). And as we have discussed, a bigger dataset size seem to have a positive correlation with improved accuracy. Despite this, we achieved strong performance, particularly given that our biggest dataset, Mixed 15 Lines, only contained 461 entries.

We also implemented strategies differently from VulRepair, focusing on limiting the number of CWE-categories and to 5 Lines changes, which could explain our performance despite having much smaller datasets. Further experimentation with limiting CWEs and Lines changes with an increased dataset size would be interesting to do in order to determine how beneficial it is to train a focused model and investigate the trade-off between a focused dataset and a large dataset. We also looked into splitting the data Randomly and on Time splits, which VulRepair does not discuss as a potential impacts on the model performance. However, we want to highlight that since VulRepair had more data samples to train on, their model might be more generalizeable compared to ours.

5.5.2 Comparison to Huang et al.'s study

The study by Huang et al. [8] is hard to compare to our own findings as they used a different dataset (SARD) and a BERT-based model. Their training set included 11,221 entries and they achieved a 95.47% Exact Match score for single-line repairs and a 90.06% Exact Match score for multi-line repairs. However, the SARD dataset consists of synthetic test cases,

which may have limited variability in method structures compared to actual commits found in open-source projects. It may even have been an attributing factor to the state-of-the-art performance they reported.

Although we cannot compare our model performance to theirs, we were inspired by their approach of limiting their dataset in terms of number of CWEs, which could have helped us gain better performance than using all available CWEs. We also learned about the BLEU score from their study, which we used alongside the Exact Match Score to evaluate our models.

Our study did not match Huang et al.'s results, but we were able to improve our models using various approaches from their study. Although we did not achieve comparable performance to theirs, we learned from their methods and developed ours further. An interesting extension of this project could be to see the effects of incorporating synthetic data from SARD in combination with real-life open-source vulnerabilities, to investigate if the additional data can contribute to a more generalizable model.

Chapter 6
Conclusion

The objective of this study was to develop a proof of concept machine learning model using the CodeT5 architecture that generates security patches. We fine-tuned our models using a dataset sourced from CVEfixes, training a total of six models. In addition, we investigated the impact of splitting the dataset by time and randomly.

Many factors can influence the performance of a machine learning model, but in our study, we found that dataset size, repository distribution, split method, CWE distribution and number of lines changed were particularly important. Interestingly, we did not observe any direct impact on model performance based solely on the programming language used. However, language choice did affect the distribution of the dataset, which made comparing the models challenging, as multiple variables varied from dataset to dataset.

Our study also revealed that there might be a time-dependecy on our dataset, causing the split method to have a significant impact on model performance. In our case, a Random split outperformed a Time split across all datasets, and in the Discussion we mentioned that the temporal aspect could be important when dealing with OSS. We want to stress that the conclusions may be entirely different, had we worked with closed-source data.

The findings from our study reveal that simply increasing the size of a dataset does not necessarily lead to improved performance, as observed in cases where the addition of more entries led to worse accuracy due to changes in the underlying distributions of the dataset. Nevertheless, these complexities can provide valuable insights into the factors that impact accuracy, both positively and negatively. Therefore, it is important to consider various factors when designing a training dataset for a specific task, as this can help develop more reliable and generalizeable models that yield better results over time. By keeping these factors in mind, we can optimize the performance of models and enhance their usefulness for various applications.

In comparing our results to prior research, we found that our model's performance was slightly lower than that of VulRepair, yet still comparable, despite our use of much smaller datasets. As for comparing our results to Huang et al.'s findings, their use of a synthetic test dataset made it challenging to compare their results directly with ours. However, we took

inspiration from their study and identified the potential benefits of limiting the number of CWE-IDs and utilizing BLEU score as a metric for evaluating model quality.

Upon comparing our findings with the related research, it became evident that there were several factors that had a significant impact on the performance of our model. Initially, our focus was limited to the EM score and BLEU score, similar to previous studies. However, upon closer examination of the dataset, we realized that there were numerous variables at play, and it was impractical to attribute the exact effect of each variable on the model's performance due to the scope of the study. Nevertheless, the experience we had provided us with valuable insights that helped us expand our understanding of the factors that impact model performance. We found that many of these factors were not addressed in the previous studies conducted by VulRepair and Huang et al. This suggests that our work could offer contributions to this area that were previously overlooked.

6.1 Future Work

Based on our project experiences, we have some identified areas that require further research. In this section, we present some suggestions for potential research ideas and improvements that may be worth exploring:

- Collecting and constructing a larger and higher-quality dataset is crucial to assess the potential for improving the model's performance and generalizability. If time dependency is a concern, it is necessary to balance the dataset with a range of entries, including both older and more recent ones. Additionally, to evaluate the model's ability to fix novel vulnerabilities effectively, a large test set containing newly dated vulnerabilities would be needed.
- Ensuring that the training data accurately reflects vulnerable code and its corresponding fixes is crucial. Designing multiple fixes for a single vulnerability may help to ensure that the training data is comprehensive and accurately reflects the code's vulnerabilities. However, we want to note that this approach would require a significant amount of manual labor and expertise in security vulnerability fixes.
- Evaluating the validity and correctness of the generated outputs through methods such as linting or static program analysis will help to determine the accuracy of the model's predictions and identify areas for improvement.
- Implementing cross-validation techniques can provide a more robust evaluation of model performance.
- Combining synthetic and real-life data to create a more robust model that can handle a wider range of inputs and outputs is another area that could be explored.
- Investigating each impacting factor in more detail could clarify how different factors influence the model's performance. For example, freezing certain variables while only examining the influence on the language or reducing CWE-categories could be useful techniques. This approach would provide a more comprehensive understanding of the model's strengths and weaknesses and help us to determine which factors have the most significant impact on its performance.

Chapter 7
Limitations and Threats to Validity

There are several limitations to our study that could potentially impact the validity of our findings. In the following section, we will discuss a few of them that we have identified.

7.1 Internal Validity

Our study was limited by the computational resources available to us. As a result, we had to make certain choices regarding hyperparameters such as batch size, number of beams and model language size. For example, we used the pre-trained model codet5-base instead of codet5-large, which we were unable to experiment with. This choice could have affected the performance of our models.

Additionally, our choices of datasets were limited, and our main objective was to create a proof of concept for Debricked. We did not manually inspect and validate the data in CVEfixes to determine whether they are all security vulnerabilities and the corresponding patches. We made assumptions about the data, including assuming that every vulnerability-patch pair in the dataset actually consists of security vulnerability fixes. However, in reality, this may not always be the case, some of the data may be non-security related, such as logs, formatting or other changes, as we discovered through brief manual inspection.

Moreover, since the files were very large, consisting of thousands of lines of code, we were unable to train on complete files including all code context. Instead, we extracted the functions that had changes, which may have included partial fi xes or non-fixes. This limitation affects the validity of our model from a security patch generation perspective. This approach was also inspired by Fu et al.[7]. Furthermore, none of the two related studies [7][8] addressed the validity of the generated code, therefore we could not determine whether they filtered out unrelated code changes or make similar assumptions as us. Although this problem might not be as applicable to Huang et al.[8] as they mainly used synthesized data.

7.2 External Validity

Our study focused on training the model on Java and Python and was restricted to five CWE categories, with datasets up to 15 lines of change within the file. However, the distribution within the CWE categories was non-uniform, and the repository distributions were also not uniform. We acknowledge that this is a significant limitation of our study, as it may limit the generalizability of our results.

The limited lines of change means that the model is biased towards shorter patches and may not be representative of real-world security patches, which can be much longer and more complex. Another important consideration is that the way in which the lines changed are extracted before processing may include comments and new lines. Therefore, the lines changed of maximum 5 or 15 does not necessarily mean code changes, but could be simply added or deleted rows, comments or similar. Furthermore, we want to note that while we attempted to pre-process the data to remove unwanted elements, there may still be some residual noise in the data, affecting the patterns that the model learns.

Another drawback was that we were not able to perform cross-validation on our training/validation/test split, due to computational resources restrictions. Instead, we only split the dataset once for every model, making the model performance tightly linked to that specific split. Having cross-validation would have minimized the dependence on randomness for the model performance and yielded more robust models.

Moreover, we only had one source for our dataset, which was open source data, and our model is most likely over-fitted to this dataset. We acknowledge that our model's results and generalizability outside of this study are limited, given the constraints of our design.

References

[1] M. Dowd, J. McDonald, and J. Schuh. The art of software security assessment: Identifying and preventing software vulnerabilities. 2007.

[2] Statistics on number of reported vulnerabilities from nvd. `https://nvd.nist.gov/vuln/search`. Accessed Jan 14, 2023.

[3] About debricked. `https://www.microfocus.com/en-us/cyberres/debricked`. Accessed Dec 29, 2022.

[4] About copilot. `https://github.com/features/copilot`. Accessed: Sep 6, 2022.

[5] About codegen. `https://github.com/salesforce/CodeGen`. Accessed: Sep 6, 2022.

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[7] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Phung Dinh. Vulrepair: A t5-based automated software vulnerability repair. *ESEC/FSE 2022: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2022.

[8] Kai Huang, Su Yang, Hongyu Sun, Chengyi Sun, Xuejun Li, and Yuqing Zhang. Repairing security vulnerabilities using pre-trained programming language models. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 111–116, 2022.

[9] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 2022.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[11] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 602–614, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019.

[13] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[15] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[16] Pawel Budzianowski and Ivan Vulic. Hello, it's GPT-2 - how can I help you? towards the use of pretrained language models for task-oriented dialogue systems. *CoRR*, abs/1907.05774, 2019.

[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[18] OpenAI. Gpt-4 technical report, 2023.

[19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[20] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859, 2021.

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[22] Bleu score explained. `https://towardsdatascience.com/foundations-of-nlp-explained-bleu-score-and-wer-metrics-1a5ba06d812b`, Accessed Feb 18, 2023.

[23] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[24] Bleu score implementation. `https://github.com/tensorflow/nmt/blob/master/nmt/scripts/bleu.py`. Accessed Feb 18, 2023.

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. pages 691–701, 05 2016.

[26] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 47(10):2162–2181, 2021.

[27] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kgenprog: A high-performance, high-extensibility and high-portability apr system. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 697–698, 2018.

[28] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM, 2021.

[29] Cvefixes database. `https://zenodo.org/record/7029359#.Y7VZUOyZP9A`. Accessed Jan 4, 2023.

[30] Example of a cve entry. `https://nvd.nist.gov/vuln/detail/CVE-2021-31245`. Accessed Mar 8, 2023.

[31] The open source definition. `https://opensource.org/osd/`. Accessed Feb 24, 2023.

[32] Patch release behaviors of software vendors in response to vulnerabilities: An empirical analysis. *Journal of Management Information Systems*, 28(4):305 – 338, 2012.

[33] Sard test suite. `https://samate.nist.gov/SARD/test-suites`. Accessed Feb 8, 2023.

[34] Vulrepair implementation. `https://github.com/awsm-research/VulRepair/blob/main/M1_VulRepair_PL-NL/vulrepair_main.py`. Accessed Feb 24, 2023.

Appendices

Appendix A
CWE Categories

This overview show the CWE categories broken down into specific CWE-IDs, grouped by View 1003.

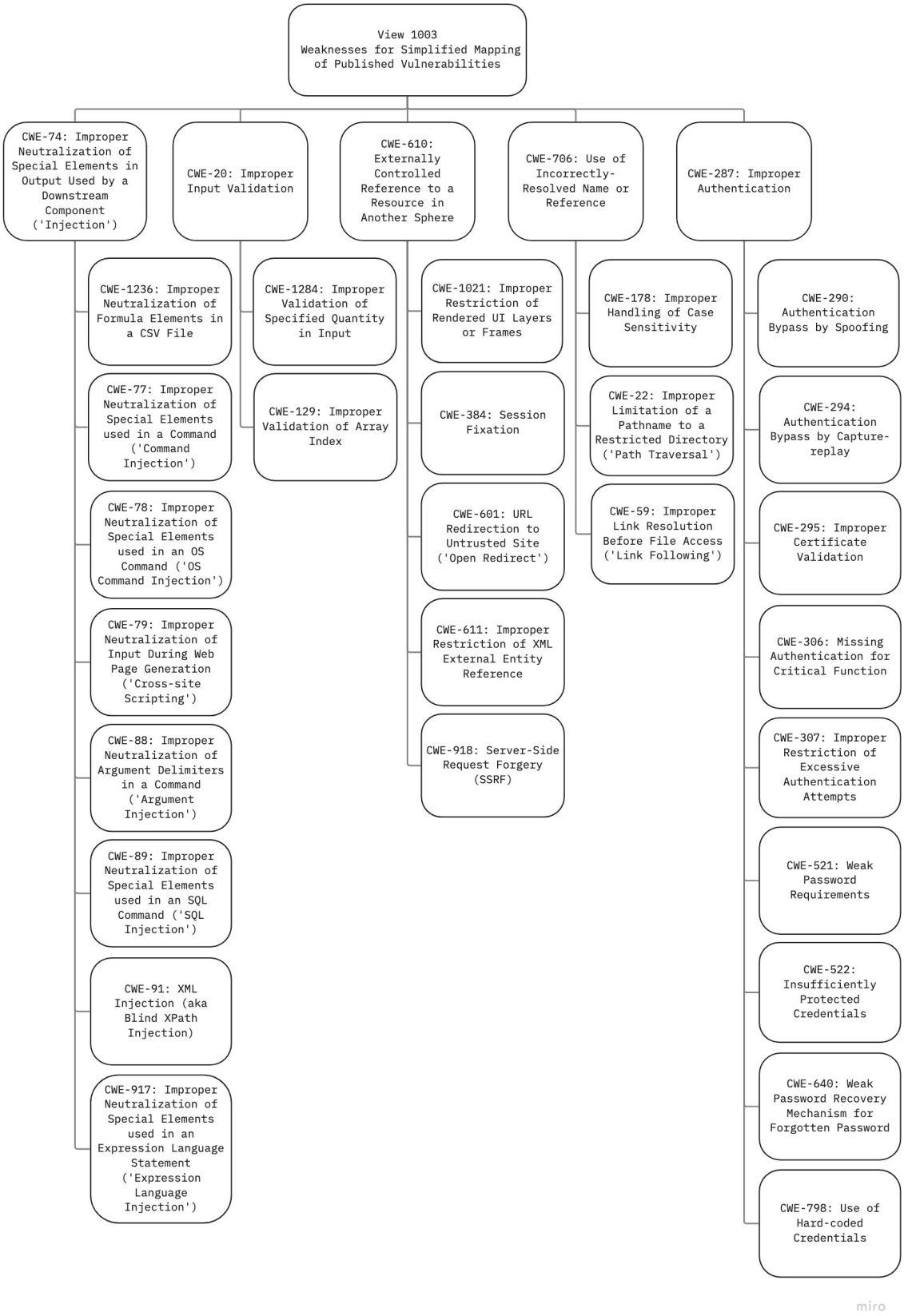

Figure A.1: Five most common CWEs in CVEfixes for Java and Python, grouped in categories.

Appendix B
More Raw Prediction Examples

Some more samples of raw predictions generated by our model. Note that the generated examples may or may not be actual security fixes as mentioned in Threats to Validity 7.

```
# Vulnerable code
CWE-74
def extra_view_dispatch(request, view):
    theme = getattr(request, "theme", None) or get_current_theme(
    request.shop)
    view_func = get_view_by_name(theme, view)

    if not view_func:
        msg = "Error! %s/%s: Not found." % (getattr(theme, "identifier"
    , None), <S2SV_StartBug> view) <S2SV_EndBug>
        return HttpResponseNotFound(msg)

    return view_func(request)

# Target and raw prediction for 5 Lines
<S2SV_ModStart> escape(view)) <S2SV_ModEnd>
```

Listing B.1: Example of correct raw predictions made by Python-5-Random

```
# Vulnerable code
CWE-74
@Override
protected HtmlRenderable htmlBody() {
  return HtmlElement.li().content(
    HtmlElement.span(HtmlAttribute.cssClass("artifact")).content(
      HtmlElement.a(HtmlAttribute.href(getUrl()))
      <S2SV_StartBug>.content(getFileName())<S2SV_EndBug>));
}

# Target and correct raw prediction for 5 Lines and 15 Lines
<S2SV_ModStart>.safecontent(getFileName())<S2SV_ModEnd>
```

Listing B.2: Example of correct raw predictions made by Java-5-Random and Java-15-Random.

```
# Vulnerable code
CWE-610
def create(request, topic_id):
    topic = get_object_or_404(Topic, pk=topic_id)
    form = FavoriteForm(user=request.user, topic=topic, data=request.
    POST)
    if form.is_valid():
        form.save()
    else:
        messages.error(request, utils.render_form_errors(form))
    return <S2SV_StartBug> redirect(request.POST.get('next', topic.
    get_absolute_url())) <S2SV_EndBug>

# Target and correct raw prediction for 5 Lines and 15 Lines
<S2SV_ModStart>
safe_redirect(request, 'next', topic.get_absolute_url(), method='POST')
<S2SV_ModEnd>
```

Listing B.3: Example of correct raw predictions made by Python-5-Random and Python-15-Random

```
# Vulnerable code
CWE-706
protected String getContent(SxSource sxSource, FilesystemExportContext
    exportContext) {
    String content;
    int originalDocParentLevel = exportContext.getDocParentLevel();
    try {
        <S2SV_StartBug> exportContext.setDocParentLevels(3); <
    S2SV_EndBug>
        content = sxSource.getContent();
    } finally {
        exportContext.setDocParentLevels(originalDocParentLevel);
    }
    return content;
}

# Target and correct raw prediction for 15 Lines
<S2SV_ModStart> exportContext.setDocParentLevels(2); <S2SV_ModEnd>
```

Listing B.4: Example of correct raw predictions made by Java-15-Time

```
# Vulnerable code
CWE-20
public void deleteById(Integer id) {
    databaseTypeDao.selectOptionalById(id).ifPresent(data -> {
        if (DatabaseTypes.has(data.getDatabaseType())) {
        throw DomainErrors.MUST_NOT_MODIFY_SYSTEM_DEFAULT_DATABASE_TYPE
    .exception();
    }
    databaseTypeDao.deleteById(id);
    <S2SV_StartBug> driverResources.delete(data.getDatabaseType()); <
    S2SV_EndBug> });
}

# Target
<S2SV_ModStart>
driverResources.deleteByDatabaseType(data.getDatabaseType());
<S2SV_ModEnd>

# Incorrect raw prediction for 5 Lines
<S2SV_ModStart> driverResources.validateJar(
data.getJdbcDriverFileTypeFileTypeFileTypeFileTypeFileTypeFileType
FileTypeFileTypeFileTypeFileTypeFileTypeFileTypeFileTypeFileType
FileTypeFileTypeFileTypeFileType());
<S2SV_ModEnd>
```

Listing B.5: Example of incorrect raw predictions made by Java-5-Time

Appendix C

Correct Predictions in Repository Distribution

The following table contains detailed information on the correct predictions for each model.

Language	Line	Split	Repository	Number of correct predictions
Java	**5**	**Random**	OpenOLAT	5
			plantuml	1
			databasir	1
			mojarra	1
			gocd	1
		Time	opencrx*	1
	15	**Random**	OpenOLAT	6
			vert.x	2
			plantuml	2
			gocd	1
			opencrx*	1
			xwiki-platform	1
		Time	OpenOLAT	3
			xwiki-platform*	1
			opencrx*	1
			plantuml	1
Python	**5**	**Random**	opendiamond	3
			shuup	3
			synapse	2
			reviewboard	1
			pysaml2	1
			bikeshed	1
			spirit	1
		Time	shuup	1
	15	**Random**	sydent	5
			spirit	3
			shuup	3
			keystone	1

Language	Line	Split	Repository	Number of correct predictions
			reviewboard	1
			recipes	1
			recurly-client-python	1
			ganga*	1
			pysaml2	1
			nova	1
		Time	sydent	6
			spirit	4
			bikeshed	1
Mixed	**5**	**Random**	OpenOLAT	5
			shuup	3
			opendiamond	3
			gocd	2
			synapse	2
			spirit	1
			bikeshed	1
			plantuml	1
			pysaml2	1
			mojarra	1
			reviewboard	1
		Time	weblate*	1
			databasir	1
			shuup	1
			opencrx*	1
	15	**Random**	OpenOLAT	6
			spirit	3
			vert.x	2
			plantuml	2
			sydent	2
			shuup	2
			xwiki-platform	1
			recurly-client-python	1
			pysaml2	1
			reviewboard	1
			gocd	1
			opencrx*	1
			keystone	1
			ganga*	1
		Time	spirit	4
			OpenOLAT	2
			sydent	1
			xwiki-platform*	1
			omero-web	1
			opendiamond*	1
			opencrx*	1

Table C.1: Correct predictions on repository distributions for all models. (*) indicate the repositories for which the correct predictions only exists in the test set.

Appendix D
Grouped Repositories

These tables details the repositories that belong to the grouped bars in Figures 4.3-4.6.

Group	Repositories
group 0	RuoYi, geotools, gocd-ldap-authentication-plugin, venice
group 1	ballerina-lang, jadx
group 2	c3p0, jbpm-wb
group 3	core, core-1, css-validator, dashbuilder, dom4j, goobi-viewer-core, hawtio, javamelody, ratpack, yaxim
group 4	dbeaver, openmrs-module-htmlformentry
group 5	drawio, opencrx
group 6	dropwizard, elasticsearch, milton2, onedev

Table D.1: Groups of Repositories for Java 5 Line

Group	Repositories
group 0	OpenClinica, xwiki-platform
group 1	RuoYi, gocd-ldap-authentication-plugin, para, pgjdbc, venice
group 2	activemq-artemis, c3p0, core-1, dbeaver, dom4j, hawtio, jbpm-wb, mpxj, portal, ratpack, shopizer
group 3	css-validator, openmrs-module-htmlformentry, spring-framework, yaxim
group 4	dashbuilder, goobi-viewer-core, javamelody, jitsi
group 5	dropwizard, elasticsearch, litemall, milton2
group 6	java-cas-client, winstone

Table D.2: Groups of Repositories for Java 15 Line

Group	Repositories
group 0	Mycodo, mayan-edms, numpy, openapi-python-client
group 1	Pillow, trape
group 2	PollBot, cpython, e2openplugin-OpenWebif, python-fedora, salt
group 3	Zope, canto-curses, keystone, mistune, moin-1.9, notebook, nova, pikepdf, pulp, pyshop, pywb, spacewalk
group 4	apkleaks, keylime, munhak-moa, openmptcprouter-vps-admin, tensorflow
group 5	archivy, web2py
group 6	askbot-devel, reviewboard
group 7	bcfg2, glances, rasa
group 8	bot, core
group 9	django-helpdesk, passhport
group 10	django-unicorn, ganga
group 11	djblets, gradio, keycloak-httpd-client-install, qutebrowser
group 12	slixmpp, thefuck

Table D.3: Groups of Repositories for Python 5 Line

Group	Repositories
group 0	Apfell, keycloak-httpd-client-install, mistune, moin-1.9, setroubleshoot
group 1	MISP, Mailpile, PollBot, Products.PluggableAuthService, cpython, django-rest-framework, e2openplugin-OpenWebif, pikepdf, pulp, python-fedora, qutebrowser, spacewalk, yum-utils
group 2	Pillow, django
group 3	Pyro3, Radicale, analytics-quarry-web, bcfg2, pyshop, thefuck
group 4	ScratchVerifier, streamlit, tensorflow
group 5	ansible, security_monkey
group 6	apkleaks, ganga, rtx
group 7	archivy, bot, django-s3file, keylime, octoprint, openmptcprouter-vps-admin, rengine, web2py
group 8	canto-curses, codecov-python, gradio, kdcproxy, python-dbusmock, wagtail
group 9	djblets, mayan-edms, numpy, openapi-python-client, passhport, pyrad, rasa, slixmpp
group 10	munhak-moa, pywb

Table D.4: Groups of Repositories for Python 15 Line

Appendix E

Distribution of Lines Changed

The following plots show a more detailed view of the Lines added and deleted before we grouped them in Figure 4.11-4.13

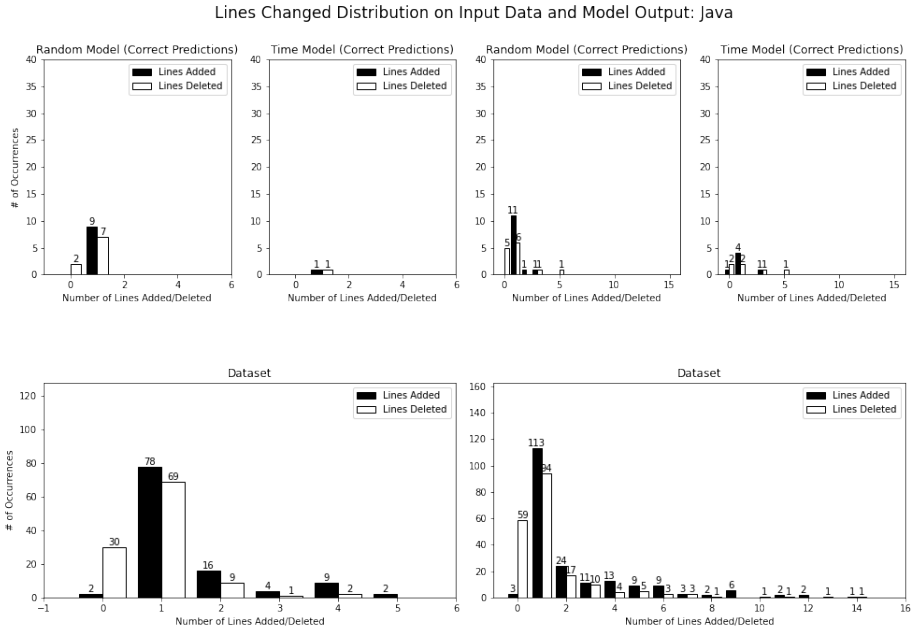

Figure E.1: Comparison of line additions and deletions in correct predictions for Java 5 Line (left plots) and Java 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

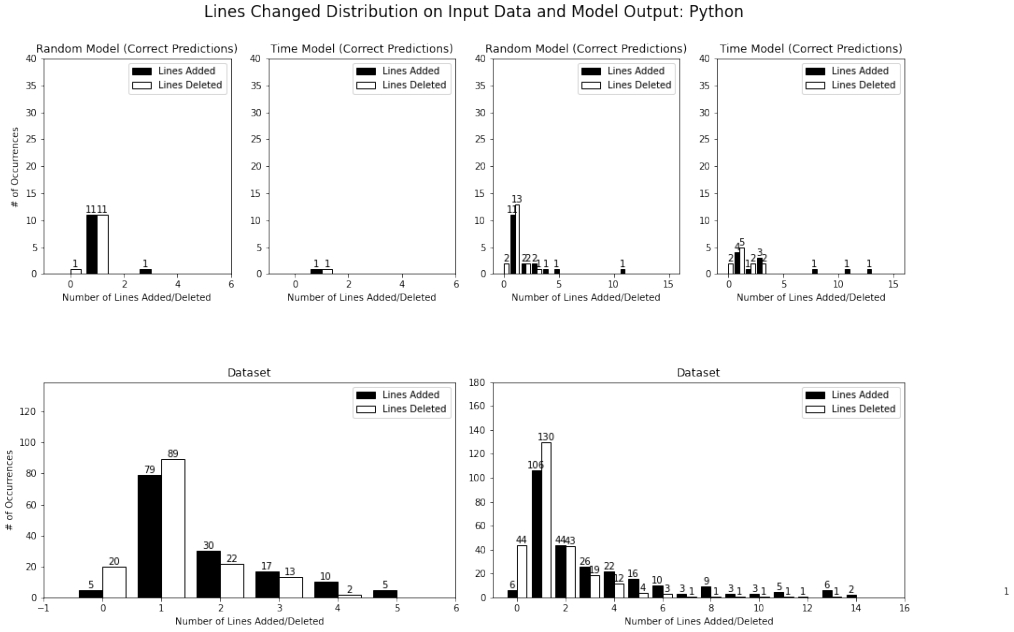

Figure E.2: Comparison of line additions and deletions in correct predictions for Python 5 Line (left plots) and Python 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

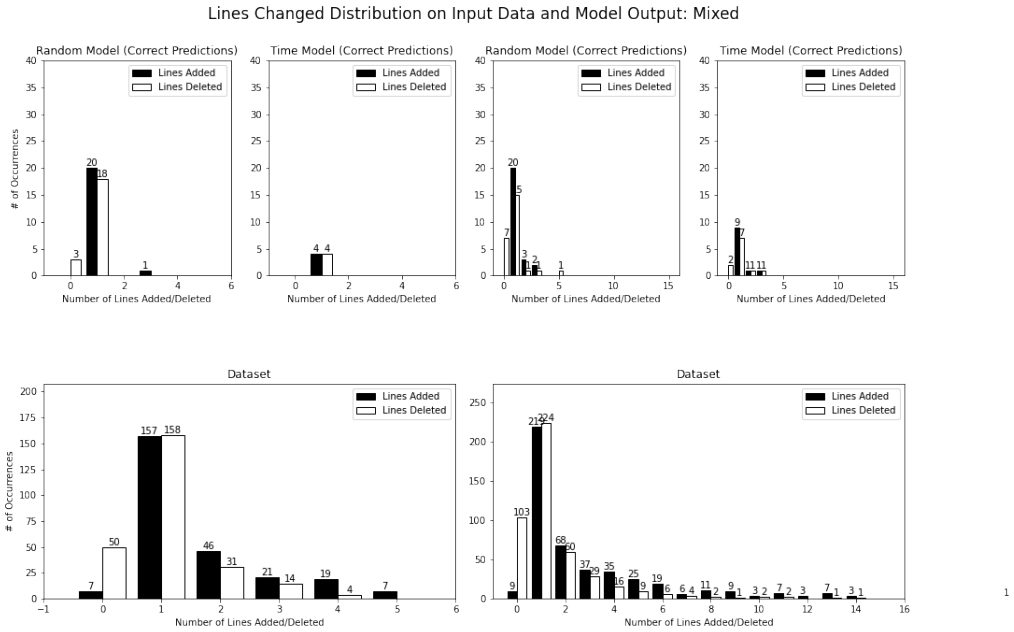

Figure E.3: Comparison of line additions and deletions in correct predictions for Mixed 5 Line (left plots) and Mixed 15 (right plots) with the Random and Time Models. The bottom diagram shows the full dataset.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-03-16

EXAMENSARBETE Maskininlärningsbaserad kodgenerering av säkerhetsfixar
STUDENTER Ewada Tsang, Cecilia Huang
HANDLEDARE Noric Couderc (LTH), Christoph Reichenbach (LTH), Emil Wåreus (Debricked)
EXAMINATOR Pierre Nugues (LTH)

Kan Artificiell Intelligens vara lösningen till automatiska säkerhetsfixar?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Ewada Tsang, Cecilia Huang**

Har du någonsin tänkt på hur många potentiella säkerhetshot du möts av varje dag? Alla dina elektroniska prylar kan bli måltavlor för angripare med onda avsikter. Från datastöld till installation av skadlig kod och mycket mer därtill, kan cyberhot sätta dig och din personliga information i fara.

Att ta itu med sårbarheter och prioritera cybersäkerhetsåtgärder är avgörande, men det kan vara skrämmande med tanke på uppgiftens komplexitet och den tid och expertis som krävs. För att bidra till forskningen inom denna utmaning, gav vi oss ut på en resa för att utforska potentialen av att använda maskininlärningsmodeller för att automatiskt generera säkerthetsfixar.

Vårt mål var att utveckla en prototyp (proof-of-concept) med hjälp av de betydande genombrotten inom Neurolingvistisk programmering (Natural Language Processing) som har gjorts de senaste åren. Forskningen har resulterat i ett stort antal modeller som är förtränade (pre-trained) för olika uppgifter, och det finns väldigt många modeller som specifikt kan förstå och generera kod. Inspirerade av relaterad forskning inom området valde vi slutligen modellen CodeT5, en språkmodell tränad på programmeringskod som är särskilt bra på både kodförståelse och kodgenerering.

Vår studie visar att det finns en stor potential inom fältet att generera säkerhetsfixar automatiskt med hjälp av maskininlärning. Vi lyckades identifiera flera intressanta insikter kring faktorer inom datasetet som vi använde för att träna modellen, vilket inte har diskuterats i tidigare forskning. Till exempel handlade det om hur vi delade upp träningsdatan inför modellträning, hur långa fixarna var, vilka projekt (repositories) som fanns i datan och sårbarhetstyperna (Common Enumeration Weaknesses). Dessa faktorer bör tas i beaktning och gör vår studie till en användbar guide för vidareutveckling av liknande modeller.

Vad gäller resultaten visar vår studie att det var möjligt att få en exakt matchning på 40% (antalet korrekta genereringar) med endast ett par hundratals samples. I kontrast lyckades en annan relaterad studie få en exakt matchning på 44%. Dock använde de ett dataset som var cirka 18 gånger större än det vi hade tillgängligt. Vi tror att ett högkvalitativt och ännu större dataset kan medföra att automatiska säkerhetsfixar definitivt kan bli en potentiell lösning inom detta fält. Vi ser fram emot att följa framtida forskning och hoppas att våra insikter kan bidra till vidareutveckling inom området.