MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Training machine learning models for vulnerability prediction and injection using datasets of vulnerability-inducing commits

DIERICKX, Jérémie

Award date:
2023

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

Training machine learning models for vulnerability prediction and injection using datasets of vulnerability-inducing commits

Dierickx Jérémie

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : Perrouin Gilles

Co-promoteur : Devroey Xavier

Mémoire présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à finalité spécialisée en Software Engineering

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Acknowledgments

With this internship I learned a lot about the machine learning field, more precisely the deep learning field, and its various applications.

I would like to thank my promoter and my co-promoter, Dr. Perrouin Gilles and Dr. Devroey Xavier from the University of Namur, who allowed me to do this thesis at the SnT of Luxembourg and helped me in its writing.

I would also like to thank Dr. Michail Papadakis and Dr. Renzo Degiovanni who helped me to develop the subject of this thesis.

I am also thankful to Dr. Aayush Garg, who helped with the concepts behind the TROVON paper and their use for this thesis.

Finally, I would like to thank all members of the Serval team, as well as the other interns present during the realisation of this thesis, for their friendliness and the good working atmosphere.

Résumé

Plusieurs techniques existent pour trouver des vulnérabilités dans du code, tel que l'analyse statique et le machine learning. Bien que les techniques de machine learning soient prometteuses, elles nécessitent une grande quantité d'exemples. Puisqu'il n'existe pas de si grande quantité de données de code vulnérable, des techniques d'injection de vulnérabilités ont été développées pour créer celles-ci. Les techniques de détection et d'injection de vulnérabilités basées sur du machine learning utilisent généralement le même type de donnée, c'est-à-dire des paires de code vulnérable, juste avant d'être corrigé, et sa version corrigée. Cependant, utiliser la version corrigée n'est pas réaliste, puisque la vulnérabilité a été introduite lors d'une autre version qui peut être bien différente par rapport à la version corrigée. Donc, nous proposons l'utilisation de paires de code ayant introduit la vulnérabilité et sa version précédente. En effet, ceci est plus réaliste, mais aussi seulement pertinent si les techniques de machine learning peuvent apprendre correctement de celles-ci et que les structures apprises sont significativement différentes qu'avec la méthode habituelle. Pour s'en assurer, nous avons entrainé des modèles de détection de vulnérabilité pour les deux types de données et comparé leurs performances. Notre analyse a démontré qu'un modèle entrainé sur des paires de code vulnérable et leur version corrigée n'est pas capable de détecter les vulnérabilités des versions ayant introduit une vulnérabilité. Il en va de même dans le sens inverse, malgré que les deux modèles sont capables d'apprendre correctement de leurs données et de détecter les vulnérabilités sur des données similaires. Donc, nous concluons que l'utilisation de codes ayant introduit une vulnérabilité pour l'entrainement de modèles de machine learning est plus pertinente que les versions corrigées.

Mots-clés *: vulnérabilité logicielle, injection de vulnérabilité, encodeur-décodeur, détection de vulnérabilité, traduction automatique*

Abstract

Multiple techniques exist to find vulnerabilities in code, such as static analysis and machine learning. Although machine learning techniques are promising, they need to learn from a large quantity of examples. Since there is not such large quantity of data for vulnerable code, vulnerability injection techniques have been developed to create them. Both vulnerability prediction and injection techniques based on machine learning usually use the same kind of data, thus pairs of vulnerable code, just before the fix, and their fixed version. However, using the fixed version is not realistic, as the vulnerability has been introduced on a different version of the code that may be way different from the fixed version. Therefore, we suggest the use of pairs of code that has introduced the vulnerability and its previous version. Indeed, this is more realistic, but this is only relevant if machine learning techniques can properly learn from it and the patterns learned are significantly different than with the usual method. To make sure of this, we trained vulnerability prediction models for both kind of data and compared their performance. Our analysis showed a model trained on pairs of vulnerable code and their fixed version is unable to predict vulnerabilities from the vulnerability introducing versions. The same goes for the opposite, despite both models are able to properly learn from their data and detect vulnerabilities on similar data. Therefore, we conclude that the use of vulnerability introducing codes for machine learning training is more relevant than the fixed versions.

Keywords *: software vulnerability, vulnerability injection, encoder-decoder, vulnerability prediction, machine translation*

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

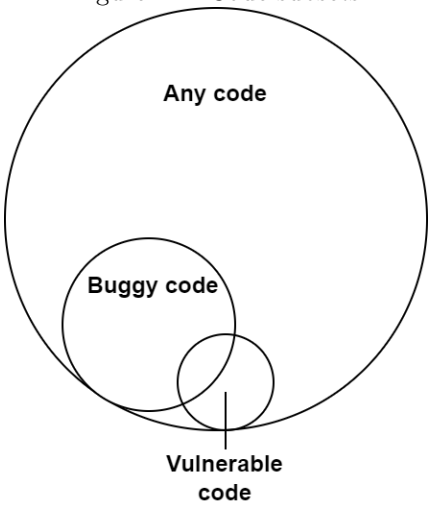
Security is an important concern in software development. While some bugs and vulnerabilities are harmless enough to be ignored, others can lead to more important repercussions like data leaks, service unavailability or even leading a system to be fully compromised.

To help organizations and their developers to detect and fix these vulnerabilities from their software, different techniques have been developed. Some of them are **software testing**, **static analysers** as well as various machine learning techniques for **vulnerability prediction**. Unfortunately, all of these techniques have their limitations and they are either time consuming or their ability to find vulnerabilities is limited.

In order to improve these techniques, we need to be able to also test them on various, realistic vulnerabilities, to measure their performances. This is one reason why vulnerability injection techniques are developed. These techniques include **genetic programming** based techniques, where the code is iteratively transformed in a way analogous to natural genetic processes, and **machine learning based techniques**, where the code transformation is based on learned patterns. [11]

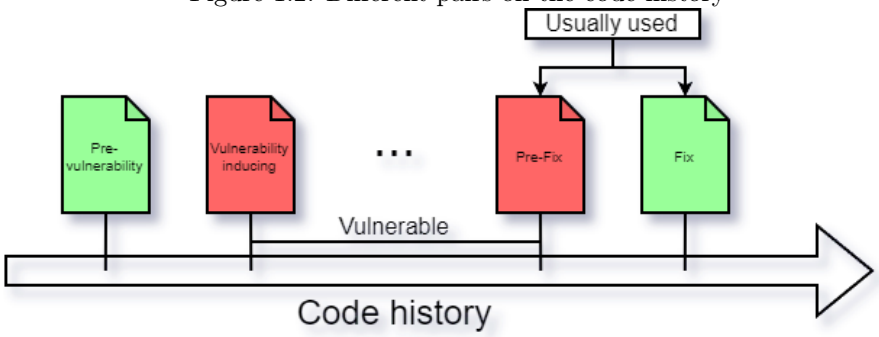
By injecting vulnerabilities into source code, we are able to create diverse samples that can be used to test vulnerability detection techniques. However, vulnerability injection also serves a second purpose that is to overcome the scarcity of vulnerability datasets for machine learning based vulnerability prediction techniques. Indeed, vulnerability prediction models suffer from a the lack of quality datasets of realistic vulnerabilities, compared to models trained on any code examples or bugs. The reason is that in reality, vulnerable code represent a very small part of any project and we are not always aware they are present in a specific program.

Figure 1.1: Code subsets



In this thesis, we focus on machine learning based vulnerability prediction and vulnerability injection techniques, that are both promising but also both suffer from this scarcity of realistic datasets. We found out that various techniques have been developed to reduce the impact of this scarcity in order to improve their models. However, datasets of pairs of pre-fix and fix are usually used for training. This kind of dataset is easier to create but is not really realistic as inducing a bug or vulnerability is usually not the same as reversing its fix.

Figure 1.2: Different pairs on the code history



1.2 Research objectives

The main objective of this thesis is to assess the use of vulnerability inducing samples, older on the code history (left side of figure 1.2), as a replacement to fixes samples (right side of figure 1.2) for training of vulnerability injection and prediction models. For that we also need to assess whether these two kind of datasets provide similar patterns to be learned by the machine learning models or not, as well as whether or not a model trained on vulnerability inducing codes could properly learn from them.

1.3 Research methodology

In order to achieve our objective, we choose to train a Seq2Seq machine learning model using a Long-Short Term Memory (LSTM) network and based on the method described

in the TROVON [7] paper. TROVON originally propose a vulnerability prediction model trained on pairs of pre-fix and fix, we will follow the same methodology but using pairs of pre-vulnerability inducing and vulnerability inducing codes.

Actually, we trained two models for each experimentation. Each model was trained on one kind of dataset (of vulnerability inducing or fixing codes) and tested on the other (of fixing or vulnerability inducing codes). In other words, each model is trained on one side of the code history (figure 1.2) and tested on the other side.

We considered two bases to create the said datasets. The first one was a dataset of manually verified vulnerability inducing commits and their corresponding fixing commits, provided with the V-SZZ [1] paper, an algorithm made to find vulnerability inducing commits from fixing commits. The second one was a dataset we made by retrieving fixing commits from the National Vulnerability Database (NVD) and using the V-SZZ algorithm on them.

Once we got our two bases for our datasets, we first split their files into functions to augment the amount of examples and reduce the complexity per example. Then, we applied a source code obfuscation algorithm on them to reduce the vocabulary size, as well as getting rid of extra information like comments.

Finally, we trained our models on these datasets to assess their ability to predict patterns of the opposite dataset to the one they were trained on.

1.4 Thesis structure

This thesis has the following structure:

- the **Background** chapter describes the state-of-the-art, tools and theoretical concepts needed in order to understand the thesis.
- the **Approach** chapter describes the different steps of our approach, the metrics we made and the parameters we choose for the trainings.
- the **Experimentation** chapter applies concretely what is explained in the Approach and adds some details to it.
- the **Results** chapter shows the different results we obtained from our experimentation with a short description.
- the **Discussion** chapter allows us to interpret the results and answering our research questions.
- the **Threats to validity** chapter lists different issues that could threaten the external or internal validity of our approach.
- the **Conclusion** chapter summarize what has been done during this thesis and suggests possible future works.

1.5 Replication package

A replication package is available for this thesis at the following github repository: `https://github.com/jdxHub/replication_package`. This package includes:

- all **datasets** in the json format.
- every notebook created during this thesis, in order to generate our datasets.
- all python scripts used to train and test our models.
- the different **models** we trained.
- graphs and tables for all metrics obtained for our models.

Chapter 2

Background

2.1 Machine Learning

State-of-the-art methods usually use pairs of vulnerable codes and their fixes, coming from popular open source projects, as this is the most accessible and reliable data that can be used for training. [15, 10, 22] Indeed, open source projects like Linux-Kernel can have a lot of publicly available fixing commits. These fixing commits are manually made and potentially verified by many developers thanks to the popularity and accessibility of these projects, so we usually make the assumption they are actually fixing the vulnerability. Though, using vulnerable codes and their fixes have some limitations as we still cannot be sure the vulnerability is accurately fixed neither if the fix did not create another vulnerability. Also, this kind of training is not really realistic as a real developer does not fix a vulnerability by simply reverting their code to an older version, so the original vulnerability injection is likely different from the injection made by reverting a fixed code to its previous version, as current methods are trained on.

2.1.1 Natural Language Processing

In machine learning, the Natural Language Processing (NPL) consists of automatically extract linguistic knowledge from a dataset of natural language texts. NPL methods are used in multiple domains like speech recognition, syntactic parsing, semantic processing, information extraction, natural language generation or machine translation. [4] Machine Translation is also useful in cases of vulnerability injection and prediction as this is able to translate a given text to another. This can be from one language to another but can also be from a first version of a text to a second, modified, version of this one (as rewording).

2.1.2 Machine Translation

There exists different approaches of Machine Translation (MT) including:

- **Rule based**. Rule-based MT relies on different levels of linguistic rules for translation, this makes use of a dictionary and a grammar, which must be developed by linguists. This approach is time-consuming and its computational cost is high [5].
- **Corpus based**. Corpus-based MT is a first alternative to overcome the knowledge acquisition problem of the rule-based approach. It automatically acquires the translation knowledge or models from bilingual corpora and is designed to work on large sizes of data [5]. The approach can be statistical (SMT), using statistical models to learn the probability of a target sentence given a source, or example-based (EBMT), using examples of similar sentence pairs from the bilingual corpus [13, 19].

  Another Corpus-based approach, the one that interests us, is the **Neural Machine Translation (NMT)**. This approach makes use of neural networks and so is a deep

learning approach to Machine Translation. The idea of NMT is to map the source into a semantic representation and then generate the translation by using attention mechanism [19]. With its ability to directly learn from the training corpora, NMT is the most used method in Machine Translation.

A Neural Machine Translation model is typically made following what is called the "Encoder-Decoder" architecture.

2.1.3 Encoder-Decoder

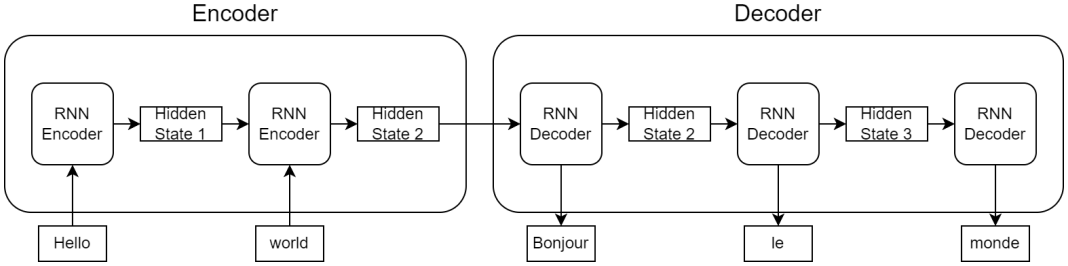
The "Encoder-Decoder" architecture is the standard neural machine translation method. The idea is to capture the context of the sequence given as input and encoding it into a "hidden state" vector first, then the decoder transforms this vector into another sequence as an output.

There are different kinds of neural networks that can be used for the encoder and decoder, as long as they are sequence based. This includes the **Recurrent Neural Networks (RNN)**, the **Gated Recurrent Units (GRU)** and the **Long-Short Term Memory (LSTM)**.

Recurrent Neural Networks (RNN)

RNN are the most basic kind of neural networks used for MT. For every element from the input sequence, the RNN encoder will apply the given element as well as the previous generated hidden state to another instance of itself to generate the next hidden state. The use of the previous generated hidden state as a second input at each iteration allows the model to remember the previous elements of the sequence and thus the context. Once the encoding is done, the last generated hidden state is passed to the RNN decoder that will produce outputs and new hidden states following the same process.

Figure 2.1: Basic Encoder-Decoder architecture using RNNs. "Hello world" translation.



Unfortunately, the RNNs face some weaknesses in its ability to remember the context in a sequence:

- RNNs suffer from "short-term memory". That means the neural network has difficulties to remember elements that are far from the current state (long-term memory). This is due to the RNN making use of gradients and two problems called the "vanishing gradient" and the "exploding gradient" problems [2].

  Even if RNN can process an infinitely large sequence thanks to its recursive aspect, what it can learn is actually very limited because of this "short-term memory" issue.

- RNNs cannot take into account the elements present **after** the current state. The basic RNNs only read each element one time and in order, so it cannot be aware of an element that comes after in this order while generating the next outputs.

  A basic alternative to overcome this is the **Bidirectional-RNN (BRNN)**, mostly consisting of processing again each element in the reverse order.

Long-Short Term Memory (LSTM)

The main difference with RNN is that, in addition to the hidden state, the LSTM also output at each iteration what is called the cell state. The cell state contains the memory from the previous iteration, allowing the neural network to remember its "short term" memory for a longer time, thus the name "**Long**-Short Term Memory" [9]. The LSTM also has another feature known as the **forget gate** that is able to decide whether or not a previous state is no longer useful and should be discarded [8].

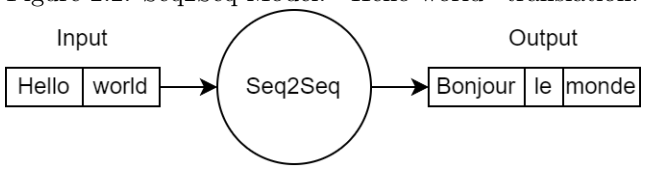
Gated Recurrent Units (GRU)

GRU networks are another alternative to the original RNNs, also aiming at adding a long term memory to it. It is similar to LSTM as it also passes the memory of the previous iteration to the current one, but this memory does not need the creation of a new state (the cell state in LSTM) and is passed in the hidden state. In addition, GRU uses what is called a **reset gate** that is able to decide also **how much** the memory from previous iterations should be forgotten.

2.1.4 Sequence-to-Sequence learning

Sequence-to-Sequence models (Seq2Seq) use an Encoder-Decoder architecture to take a sequence of items and gives another sequence of items as an output. This is this type of model we will train in this thesis, with LSTM encoders and decoders. Again, the most intuitive use case of this kind of models is for natural language translation, where it could take a sequence of words in English as an input and outputs a sequence of these words translated in French.

Figure 2.2: Seq2Seq Model. "Hello world" translation.



In addition to the simple Encoder-Decoder architecture, a Seq2Seq model also comes with some new features: the **Copy mechanism**, the **Attention mechanism** and the **Beam search**.

Copy mechanism

When training a NMT model with the Encoder-Decoder architecture, your training data contains a certain vocabulary that the model is trained on. Unfortunately, this training data is unlikely to contain all the possible vocabulary and should not (until necessary), as the bigger is the vocabulary, the more there is to learn and there is a performance decrease. That means the model can have to translate a word it does not know, that is where the **Copy mechanism** is used. The Seq2Seq model will copy this word it does not know and use it in its output.

Attention mechanism

In the traditional Encoder-Decoder architecture, only the last hidden state from the encoder is passed to the decoder and so it has to predict the output sequence from only this last hidden state. Even if LSTM and GRU exist to overcome the short term memory issue, they still have limitations when facing long sequences. The **Attention mechanism** helps to overcome this limitation by passing not only the last hidden state to the decoder but all of them. Allowing it to focus on different parts of the input sequence as needed.

Beam search

The Beam search is a heuristic algorithm consisting in letting the decoder consider not only the prediction with the best score but the k top predictions. At each iteration, the decoder tries to make its predictions based on the k previous top predictions and outputs the next k top predictions. This algorithm helps at finding better outputs at the cost of more computational resources. The size k of the beam search is usually chosen based on the vocabulary size and the sequence length for that reason.

2.1.5 Machine learning weaknesses

Some recent papers like VULGEN [15] are trying to overcome the weak ability of machine learning models to understand code semantically, when training on small datasets, by combining it with other, deterministic, methods such as pattern mining. Indeed, machine learning methods are probabilistic and require a large quantity of quality data to be trained on to be reliable. Since we do not have that much data for vulnerability injection and vulnerability detection, machine learning methods are still performing poorly in this domain compared to other domains where we can train it on more data. To overcome this weakness, VULGEN only uses a CodeT5 [21] model fine-tuned on vulnerabilities to determine *where* to inject the vulnerability and then use a deterministic method (pattern mining) to effectively inject the vulnerability. However, all these methods still use pairs of pre-fixed and fixed code to train their models.

2.1.6 TROVON

TROVON [7] is a method consisting of training a Seq2Seq model to *identify* vulnerable code and not to *fix* it despite Seq2Seq models being able to generate an output sequence from the input sequence. The reason is again the scarcity of quality data for vulnerabilities. TROVON assert that information gained is inevitably incomplete and thus the Seq2Seq is not reliable. However, this output sequence can still be used to indicate the presence of a vulnerability. Indeed, in cases where the model modifies the input, TROVON considers the input code was vulnerable, otherwise they are non-vulnerable. For the model to be as reliable as possible for this purpose, the dataset is made by using pairs of vulnerable files and their fix, splitting them into pairs of functions and abstracting them to reduce the vocabulary. The result is a dataset of pairs of vulnerable functions and their fix but also of unmodified, assumed non-vulnerable, functions. The abstraction allowing the model to not taking into account this like variable and function names, that are unnecessary information in this context.

This is this method that we will use for our comparison. As vulnerability inducing code suffer from the same scarcity as the fixing ones, this would be irrelevant to compare the exact generated, inevitably incomplete, outputs. The TROVON method seems to be best suited for our comparison as it allows us to only compare vulnerability patterns learned by both models (fixes based and vulnerability inducing base).

2.2 SZZ algorithm

The SZZ algorithm is usually used to identify changes that are likely to introduce bugs. For this purpose, an SZZ algorithm relies on historical data and tracking systems to identify changes in source code. More precisely, it locates previous changes of the lines modified in a fixing commit and the last ones made before any bug report are flagged as being bug-inducing. [16]

For this purpose, an SZZ algorithm uses different tools, such as git (or any other **version control system**), as well as a **diff tool** and a **blame tool**.

- As the name suggests, a **diff tool** is able to recognise differences between two inputs. It allows the SZZ algorithm to know what has been modified between two commits.
- A **blame tool** is able to identify all previous commits that last modified each line of a file. An example of blame tool is the *git blame* command [3].

The accuracy of SZZ implementations is usually low and one of the causes is that the SZZ algorithm is usually sensitive to refactoring changes, that should not be considered as bug-inducing. The Refactoring Aware SZZ Implementation (RA-SZZ [14]) aims at fixing this weakness, slightly improving the accuracy.

Since there is no existing dataset of vulnerability inducing commits, a refactoring aware implementation of the SZZ algorithm, specifically aiming at finding **vulnerability-inducing** commits and called V-SZZ, will be used in our approach.

Chapter 3

Approach

With this approach, we will try to answer the two following research questions during our experimentation.

RQ1.Are patterns learned by both models similar?

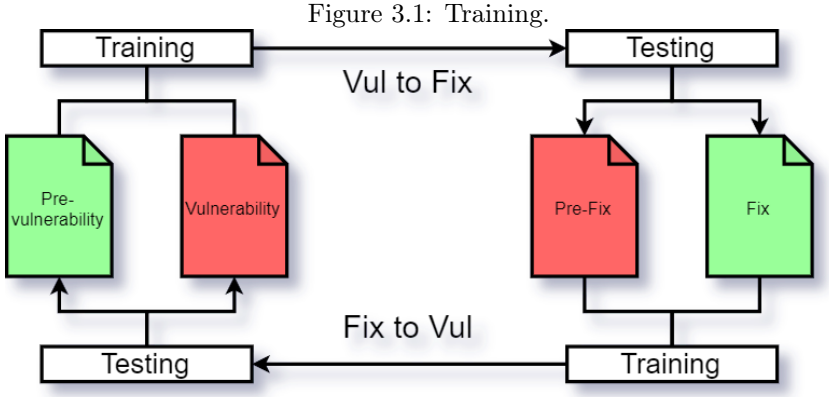
If patterns are similar, that means there is near to no point to use vulnerability inducing codes for training. We try to determine whether or not patterns are similar by comparing models performance on each other dataset. A similar performance for both models could indicate patterns learned are similar, a model performing significantly better than the other would mean patterns are different.

RQ2.Are vulnerability inducing codes suitable for training?

We try to assess whether or not a Seq2Seq model can learn efficiently from our vulnerability inducing datasets. If our models do not learn correctly from it, it could mean that this kind of training is not suitable or that our approach to extract and label the data has to be improved. We could also consider the vulnerability inducing code is not suitable for training in case models trained on fixes perform a lot better.

To answer these, we are training a vulnerability prediction model using pairs of vulnerability inducing commits and their previous commits as its data and testing it on the corresponding vulnerable-fixed pairs. We also train another model on the vulnerable-fixed pairs, as usually done, and test it on the pairs of vulnerability inducing code and their previous version.

Figure 3.1: Training.



To summarize, for our approach we need to:

- get or create **two datasets**. One is composed of fixing commits and their corresponding previous commits (Pre-fix/Fix pairs), the other is composed of vulnerability inducing commits and their corresponding previous commits (Pre-vulnerability inducing/vulnerability inducing pairs). Later we will consider these two datasets are one composed of both pairs.
- augment the dataset by **splitting files into functions**. It means that instead of having the original code containing multiple functions in a single file, we have a different file for each function. Thanks to this, we have a lot more, smaller, examples for training and testing.
- **label** the functions as vulnerable or not vulnerable.
- **abstract** the functions to reduce the vocabulary size.
- train the two Seq2Seq models using a LSTM Encoder-Decoder.
- evaluate the performances of both models on the other one's dataset.

3.1 Used datasets

For this approach, we made three different datasets of different sizes.

The **smallest one** is composed of vulnerability inducing commits and the corresponding fixing commits. These commits have been extracted from the dataset of manually verified inducing commits from the V-SZZ paper [1]. This dataset contains a total of 1492 functions for its inducing commits part and 3275 functions for the fixing commits one. We have more confidence in the quality of this dataset as its content has been verified manually.

The **largest one** where fixing commits have been extracted from the National Vulnerability Database (NVD) then the vulnerability inducing commits have been found using the V-SZZ algorithm. This dataset is made using these vulnerability inducing commits and contains a total of 104.588 functions, including 6051 vulnerable ones. We have less confidence in the quality of this dataset, as the V-SZZ has a low accuracy according to the V-SZZ paper (around 60%).

Finally, the **third one** has been made because we could not train the largest one because of a lack of resource and time. This dataset is a reduced version of the largest one and contains a total of 13073 functions including 756 vulnerable ones.

All datasets were then processed following the TROVON method described in 2.1.6. We firstly used **SrcML** [12] to decompose files into functions. Then we applied **src2abs** [17, 18] on these functions to abstract them and reduce the vocabulary size. We finally labelled them as the following. All functions that have been modified from the vulnerability inducing commit or by the fixing commit are considered as vulnerable. The others are Non-vulnerable.

3.2 Splitting into functions

As mentioned before, each file of our datasets has been split into functions. There are two reasons to this choice:

- **Augmenting** the datasets. Since there is not many examples of realistic pairs of vulnerable and non-vulnerable code files, splitting them into functions is an easy way to greatly increase the amount of different pairs.

- **Reducing the sequence length** of our data. Because Seq2Seq models still work better on smaller sequence length, and also because a bigger sequence length needs more memory (RAM) to be trained on. On larger datasets, we will need to reduce the maximum sequence length to reduce the memory usage. That means our data will be truncated, leading to information loss. The bigger is the sequence length of our data, the greater is the loss once truncated.

3.3 Abstracting the code

Each function of our datasets is abstracted. That means we replace some words (variable names, class names, ...) by other, generic ones. It allows us to **reduce the vocabulary** size, leading to the two following benefits:

- a better **performance** of the model. As the model is trained on a smaller vocabulary, mostly the same for every example, it can better learn the semantic of our data rather than the syntax.As an example, training a model with the original variable names could lead the model to link some variable names to specific vulnerabilities, thus its predictions could depend on the variable names in the input rather than the semantic.
- a smaller **memory** usage. As the model has fewer words to map.

Because some function pairs could call their variables in a different order or having a different number of variables, we abstract our functions by pairs. That means the abstraction is mapped for both functions at the same time, so two functions (in a pair) with one of these differences will keep this difference in their abstracted versions.

Here is an example with the **ext4_ext_try_to_merge** function from the **Linux Kernel** repository:

- First we have a version of ext4_ext_try_to_merge coming from a **vulnerability inducing** commit.

```
static int ext4_ext_try_to_merge(struct inode *inode,
             struct ext4_ext_path *path,
             struct ext4_extent *ex) {
   struct ext4_extent_header *eh;
   unsigned int depth;
   int merge_done = 0;
   int ret = 0;

   depth = ext_depth(inode);
   BUG_ON(path[depth].p_hdr == NULL);
   eh = path[depth].p_hdr;

   if (ex > EXT_FIRST_EXTENT(eh))
      merge_done = ext4_ext_try_to_merge_right(inode, path, ex - 1);

   if (!merge_done)
      ret = ext4_ext_try_to_merge_right(inode, path, ex);

   return ret;
}
```

- Then we have a version of ext4_ext_try_to_merge coming from the **previous** commit to the vulnerability inducing one.

```
static int ext4_ext_try_to_merge(struct inode *inode,
            struct ext4_ext_path *path,
            struct ext4_extent *ex)
{
   struct ext4_extent_header *eh;
   unsigned int depth, len;
   int merge_done = 0;
   int uninitialized = 0;

   depth = ext_depth(inode);
   BUG_ON(path[depth].p_hdr == NULL);
   eh = path[depth].p_hdr;

   while (ex < EXT_LAST_EXTENT(eh)) {
      if (!ext4_can_extents_be_merged(inode, ex, ex + 1))
         break;
      /* merge with next extent! */
      if (ext4_ext_is_uninitialized(ex))
         uninitialized = 1;
      ex->ee_len = cpu_to_le16(ext4_ext_get_actual_len(ex)
            + ext4_ext_get_actual_len(ex + 1));
      if (uninitialized)
         ext4_ext_mark_uninitialized(ex);

      if (ex + 1 < EXT_LAST_EXTENT(eh)) {
         len = (EXT_LAST_EXTENT(eh) - ex - 1)
            * sizeof(struct ext4_extent);
         memmove(ex + 1, ex + 2, len);
      }
      le16_add_cpu(&eh->eh_entries, -1);
      merge_done = 1;
      WARN_ON(eh->eh_entries == 0);
      if (!eh->eh_entries)
         EXT4_ERROR_INODE(inode, "eh->eh_entries = 0!");
   }

   return merge_done;
}
```

We can see there are a lot of changes. Among these changes, we can notice both versions start with different variables. By **abstracting by pairs**, the results are the following.

- First, the abstracted version of ext4_ext_try_to_merge coming from a **vulnerability inducing** commit.

```
static int VAR_1 ( VAR_2 VAR_3 * VAR_3 , VAR_2 VAR_4 * path , VAR_2 VAR_5 *
    VAR_6 ) { VAR_2 VAR_7 * VAR_8 ; VAR_9 int VAR_10 ; int VAR_11 = 0 ; int
    VAR_12 = 0 ; VAR_10 = VAR_13 ( VAR_3 ) ; VAR_14 ( path [ VAR_10 ] .
    VAR_15 == NULL ) ; VAR_8 = path [ VAR_10 ] . VAR_15 ; if ( VAR_6 > VAR_16
    ( VAR_8 ) ) VAR_11 = VAR_17 ( VAR_3 , path , VAR_6 - 1 ) ; if ( ! VAR_11
    ) VAR_12 = VAR_17 ( VAR_3 , path , VAR_6 ) ; return VAR_12 ; }
```

- Then the abstracted version of ext4_ext_try_to_merge coming from the **previous** commit to the vulnerability inducing one.

```
static int VAR_1 ( VAR_2 VAR_3 * VAR_3 , VAR_2 VAR_4 * path , VAR_2 VAR_5 *
    VAR_6 ) { VAR_2 VAR_7 * VAR_8 ; VAR_9 int VAR_10 , VAR_18 ; int VAR_11 =
    0 ; int VAR_19 = 0 ; VAR_10 = VAR_13 ( VAR_3 ) ; VAR_14 ( path [ VAR_10 ]
    . VAR_15 == NULL ) ; VAR_8 = path [ VAR_10 ] . VAR_15 ; while ( VAR_6 <
```

```
VAR_20 ( VAR_8 ) ) { if ( ! VAR_21 ( VAR_3 , VAR_6 , VAR_6 + 1 ) ) break
; if ( VAR_22 ( VAR_6 ) ) VAR_19 = 1 ; VAR_6 - > VAR_23 = VAR_24 ( VAR_25
( VAR_6 ) + VAR_25 ( VAR_6 + 1 ) ) ; if ( VAR_19 ) VAR_26 ( VAR_6 ) ; if
( VAR_6 + 1 < VAR_20 ( VAR_8 ) ) { VAR_18 = ( VAR_20 ( VAR_8 ) - VAR_6 -
1 ) * VAR_27 ( VAR_2 VAR_5 ) ; VAR_28 ( VAR_6 + 1 , VAR_6 + INT_1 ,
VAR_18 ) ; } VAR_29 ( & VAR_8 - > VAR_30 , - 1 ) ; VAR_11 = 1 ; VAR_31 (
VAR_8 - > VAR_30 == 0 ) ; if ( ! VAR_8 - > VAR_30 ) VAR_32 ( VAR_3 ,
STRING_1 ) ; } return VAR_11 ; }
```

For a better visualization, here is also the original and abstracted versions with highlighted differences.

Figure 3.2: Differences in a pair of functions

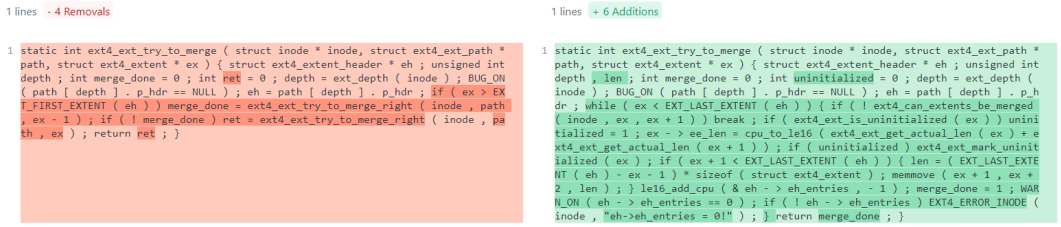
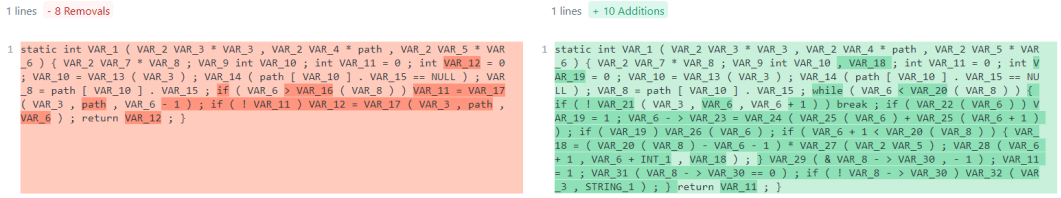


Figure 3.3: Differences in a pair of functions after abstraction by pairs



We can see that the differences between the two functions in this pair are preserved. Names that are the same between the original versions are also the same between the abstracted versions. We can also notice that the abstraction discards the formatting to have everything in one line, as well as the comments that are no longer present in the abstracted version.

In comparison, here are the results when abstracting the two versions separately.

- First, the abstracted version of ext4_ext_try_to_merge coming from a **vulnerability inducing** commit.

```
static int VAR_1 ( VAR_2 VAR_3 * VAR_3 , VAR_2 VAR_4 * path , VAR_2 VAR_5 *
    VAR_6 ) { VAR_2 VAR_7 * VAR_8 ; VAR_9 int VAR_10 ; int VAR_11 = 0 ; int
    VAR_12 = 0 ; VAR_10 = VAR_13 ( VAR_3 ) ; VAR_14 ( path [ VAR_10 ] .
    VAR_15 == VAR_16 ) ; VAR_8 = path [ VAR_10 ] . VAR_15 ; if ( VAR_6 >
    VAR_17 ( VAR_8 ) ) VAR_11 = VAR_18 ( VAR_3 , path , VAR_6 - 1 ) ; if ( !
    VAR_11 ) VAR_12 = VAR_18 ( VAR_3 , path , VAR_6 ) ; return VAR_12 ; }
```

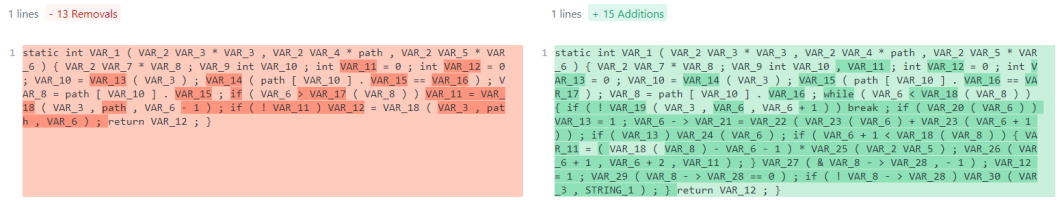
- Then the abstracted version of ext4_ext_try_to_merge coming from the **previous** commit to the vulnerability inducing one.

```
static int VAR_1 ( VAR_2 VAR_3 * VAR_3 , VAR_2 VAR_4 * path , VAR_2 VAR_5 *
    VAR_6 ) { VAR_2 VAR_7 * VAR_8 ; VAR_9 int VAR_10 , VAR_11 ; int VAR_12 =
    0 ; int VAR_13 = 0 ; VAR_10 = VAR_14 ( VAR_3 ) ; VAR_15 ( path [ VAR_10 ]
    . VAR_16 == VAR_17 ) ; VAR_8 = path [ VAR_10 ] . VAR_16 ; while ( VAR_6 <
```

```
VAR_18 ( VAR_8 ) ) { if ( ! VAR_19 ( VAR_3 , VAR_6 , VAR_6 + 1 ) ) break
; if ( VAR_20 ( VAR_6 ) ) VAR_13 = 1 ; VAR_6 - > VAR_21 = VAR_22 ( VAR_23
( VAR_6 ) + VAR_23 ( VAR_6 + 1 ) ) ; if ( VAR_13 ) VAR_24 ( VAR_6 ) ; if
( VAR_6 + 1 < VAR_18 ( VAR_8 ) ) { VAR_11 = ( VAR_18 ( VAR_8 ) - VAR_6 -
1 ) * VAR_25 ( VAR_2 VAR_5 ) ; VAR_26 ( VAR_6 + 1 , VAR_6 + 2 , VAR_11 )
; } VAR_27 ( & VAR_8 - > VAR_28 , - 1 ) ; VAR_12 = 1 ; VAR_29 ( VAR_8 - >
VAR_28 == 0 ) ; if ( ! VAR_8 - > VAR_28 ) VAR_30 ( VAR_3 , STRING_1 ) ; }
return VAR_12 ; }
```

Here is the abstraction with highlighted differences.

Figure 3.4: Differences in a pair of functions after a simple abstraction



We can see that each new variable names are incremented following their apparition order in each function separately, leading the differences between the two to be lost information.

3.4 Labelling

As we only have two possibilities, vulnerable and non-vulnerable, we have a binary classification with **0** corresponding to a non-vulnerable function and **1** corresponding to a vulnerable one. As for how we consider a function is vulnerable or not during labelling, we set as **vulnerable** every function that were modified between two files of the same pair. The rest is left as **non-vulnerable**.

3.5 Used metrics

When testing our models, we extract different metrics to evaluate the performances. Here we explain these different metrics and what is calculated. First, the most basic metrics.

Computations of the others are made based on these:

- True Negatives **(TN)**. This is the number of predicted non-vulnerable code that are also actually labelled as non-vulnerable in our testing dataset. The prediction is correct.
- False Negatives **(FN)**. This is the number of predicted non-vulnerable code that are actually labelled as **vulnerable** in our testing dataset. The prediction is incorrect.
- True Positives **(TP)**. This is the number of predicted vulnerable code that are also actually labelled as vulnerable in our testing dataset. The prediction is correct.
- False Positives **(FP)**. This is the number of predicted vulnerable code that are actually labelled as **non-vulnerable** in our testing dataset. The prediction is incorrect.

Second, the more relevant metrics.

- **Accuracy**

$$\frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \iff \frac{TP+TN}{TP+TN+FP+FN} \tag{3.1}$$

- **Precision**. This is the proportion of correct positive predictions.

$$\frac{TP}{TP + FP} \tag{3.2}$$

- **Recall**. This is the proportion of correct positives that have been correctly predicted as positive.

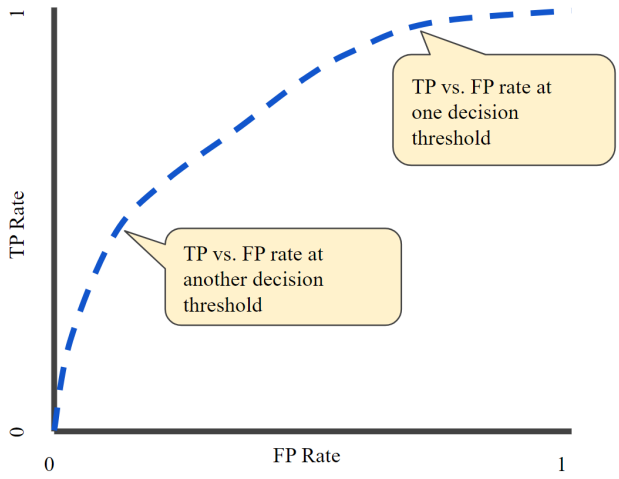
$$\frac{TP}{TP + FN} \tag{3.3}$$

- **F-measure**. This metric combines Precision and Recall into a single value by calculating their harmonic mean.

$$\frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.4}$$

- **AUC**. This is a metric based on the **ROC curve (receiver operating characteristic curve)** [6]. The ROC curve plots the **True Positive Rate** (same as the Recall) and the **False Positive Rate** ($\frac{FP}{FP+TN}$) at different thresholds of classification, as in the following figure.

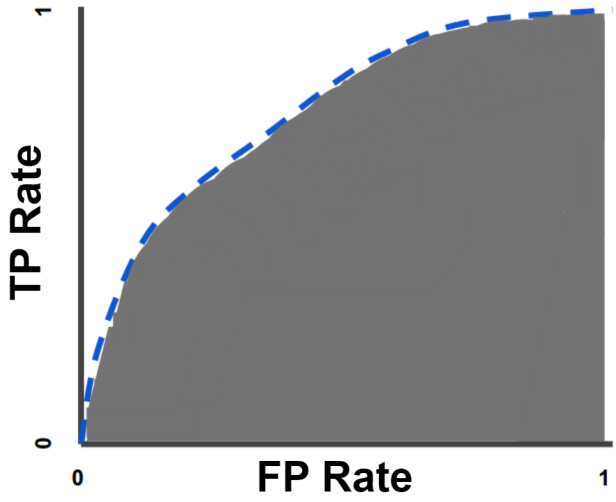
Figure 3.5: ROC curve[1]



AUC stands for **Area under the ROC Curve**. That means this is the measure of this area as shown in the following figure.

[1] https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

Figure 3.6: Area under the ROC Curve



The bigger is this area, the better it is, and the AUC score comes close to 1.

- **Precision-Recall AUC**. This metric is the same as the AUC, except the curve is based on Recall and Precision instead of Recall and the False Positive Rate. This metric works better than the original AUC for heavily imbalanced datasets (with a high amount of negative instances) as it focuses more on positive instances.

- **Matthews Correlation Coefficient (MCC)**. This metric exists because accuracy is a less relevant metric when a label have (a lot) more instances than another (we have heavily imbalanced data).

  The MCC is calculated as the following:

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.5}$$

  The resulting value varies between -1 and 1. A score near to -1 means terrible performances, a score of 0 means that the model is no better than random and a score of 1 indicates a perfect match.

- **Loss**. The loss indicates, during training, how bad was the model prediction on a single example from the training dataset. Also considered as a "penalty" for a bad prediction, that means the model should be better when close to 0. There exists various loss functions to evaluate this score for different kinds of problems. In case of classification problems, some are cross-entropy, logarithmic, exponential or pinball. [20]

- **Validation Loss (Val Loss)**. This is the same as the (Training) Loss but from bad predictions of the validation dataset. The validation dataset is a portion of our dataset that is not used to train the model. That means the Validation Loss indicates the performance of the model on new data.

3.6 Training

To have an idea of whether or not the model is overfitting during the training and try to avoid it, the dataset originally used for the training is split into a training dataset and a validation dataset.

The training dataset is effectively used for the training while the validation dataset is smaller and contains functions that are unknown to the model and are of the same category (Pre-Fix/Fix or Pre-Vulnerability/Vulnerability inducing) as the training dataset.

This way we can keep track of the model performance during the training using both the **Loss** (on known data) and the **Validation Loss** (on unknown data).

3.6.1 Dataset distribution

All models have been trained on their corresponding dataset with the following distribution:

- **90%** of the dataset is used for **training**. Since training properly a model usually requires a lot of data, we need to keep as much data as possible for the training dataset over the validation dataset.
- **10%** of the dataset is used for **validation**. We do not need as much data as for training, since the validation dataset is used to quickly validate the model on unknown data **during** training and calculate the **Validation Loss**.
- The **testing dataset** corresponds to the whole dataset of the opposite category (Pre-Fix/Fix or Pre-Vulnerability/Vulnerability inducing) to the training and validation dataset.

3.6.2 Hyperparameters

The hyperparameters used for every training are the following and are the same as in the original TROVON repository.

- Batch size: 64
- Num heads: 8
- Separator: "\t"
- Reserved tokens: [PAD], [UNK], [START], [END]

The only difference between models is that models trained on the small dataset had a Maximum Sequence Length of 900 and the model trained on the medium dataset had a Sequence Length of 100 as for the original TROVON model.

Chapter 4

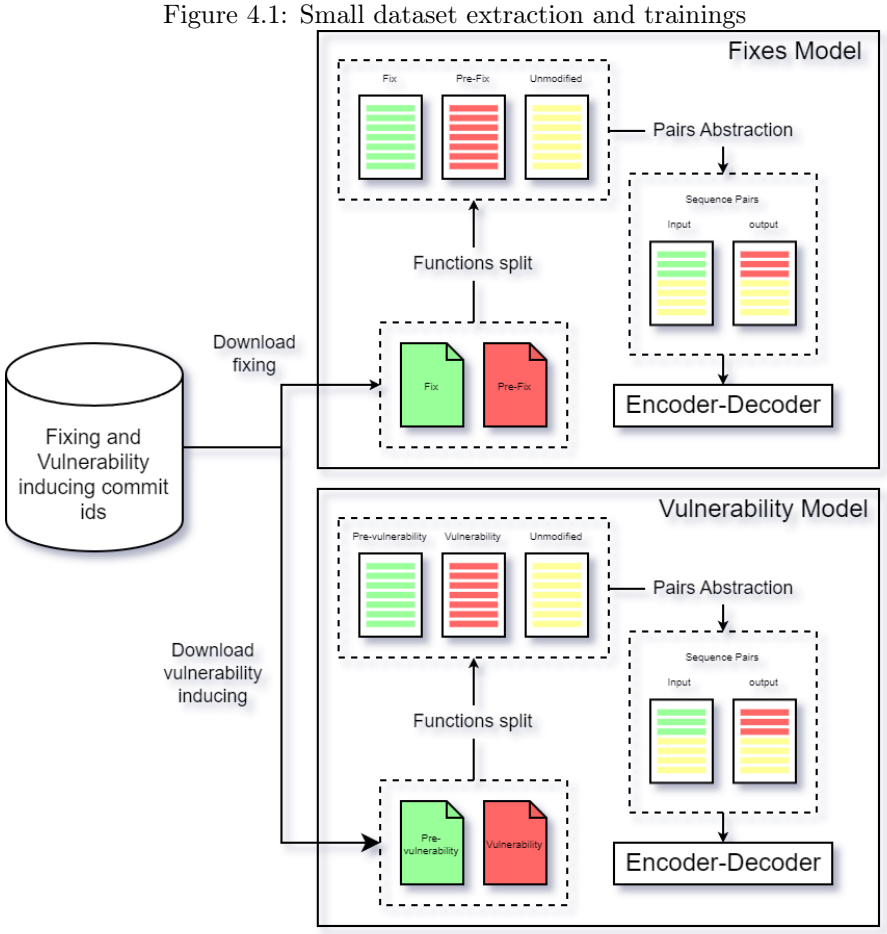
Experimentation

All experiments have been made on **google colab** notebooks, allowing us to use GPU acceleration and run them at all time. It also makes the replication easier as every instance of google colab are on the same operating system and mostly the same hardware.

4.1 Small dataset

The small dataset has been made using a dataset of manually verified pairs of fixing commits and their corresponding vulnerability inducing commits on multiple projects. We made a python script to extract the commit ids from this dataset and download the modified files from their repositories using the Git and PyDriller libraries for python. Once all files are downloaded, we split them into functions as explained in the approach (chapter 3) before training our Seq2Seq model on them. We did that twice as we need a model trained on fixes and another one trained on vulnerability inducing functions, as shown on the following figure.

Figure 4.1: Small dataset extraction and trainings



Multiple pairs of models have been trained on this dataset for two main reasons:

- as the dataset is very small, it may not generalize well on new data. So we wanted to get the best possible evaluation out of it by doing a cross-validation.
- it was very quick to be trained on, even multiple times, thanks to its small size.

A training was made on a shuffled version of the dataset and the rest was made in 10 folds, for both the vulnerability-inducing to fixes model and the fixes to vulnerability-inducing model.

Here is the distribution of functions per project for the small dataset.

Type	ImageMagick	FFmpeg	Linux-kernel	OpenSSL	Php-src
Inducing	45	526	516	200	205
Fixes	227	769	1404	429	446
Total	272	1295	1920	629	651

4.1.1 Shuffled

We randomly shuffle the dataset so that the model is not trained on an ordered dataset. This helps to minimise variance and ensures the model is not overfitting to certain patterns. To make sure we had both vulnerable and non-vulnerable samples in both training and validation parts, we chose to do the following steps:

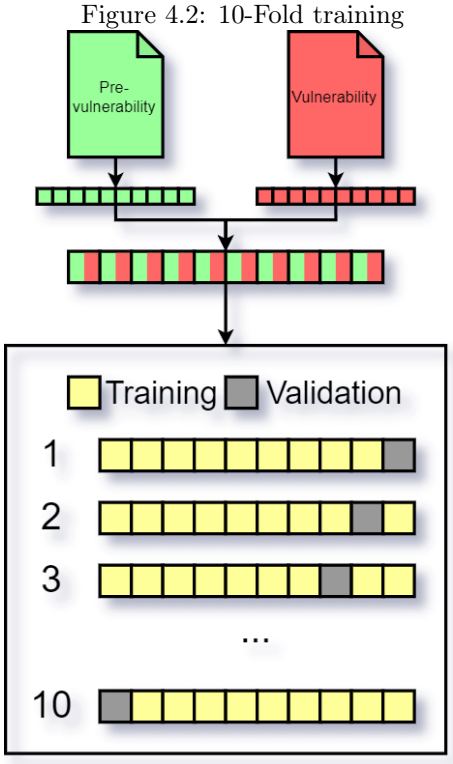
- We split the dataset into two parts, one with vulnerable code only and the other with non-vulnerable code.
- We shuffled the two datasets and made a training set and a validation set for both of them separately.
- We merged the two training sets and the two validation sets into one of each.

4.1.2 10-Fold

The K-Fold Cross-Validation allows us to better evaluate a machine learning model for a limited dataset. The dataset is split into K groups, allowing us to train K models using K-1 groups for the training data and 1 group for the validation. Once all the K models have been trained, we can compare and evaluate them. Here we trained on a 10-Fold Cross-Validation and so we trained 10 models.

As for the shuffled version, to make sure we keep the same portion of vulnerable code in training data and validation data for each model, we first split the original into two, a dataset of non-vulnerable code and another of vulnerable code, we split each of them into K groups then we merged each group of non-vulnerable code with the matching group of vulnerable code. The following figure represents this.

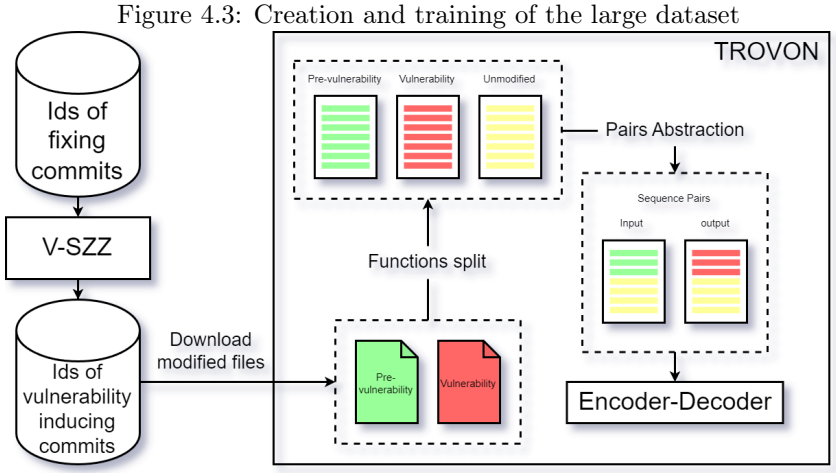
Figure 4.2: 10-Fold training



4.2 Medium dataset

The medium dataset is a reduced version of the large dataset we made as, with the free tier hardware provided by google colab, we could not train on this last one. That means the method used to create the large dataset also applies to this one and is represented by the following figure.

Figure 4.3: Creation and training of the large dataset



For this dataset we only trained one model, not shuffled, as we no longer had the time to make more for this thesis. Also both the large and medium datasets contain code from the **Linux kernel** repository only.

Chapter 5

Results

5.1 Small dataset

Models were trained on the small dataset with the following settings.

Type	Training pairs (90%)	Validation pairs (10%)	Vulnerable
Inducing	1342	150	160
Fixes	2946	329	101

Vocabulary size	Sequence length
1301	900

5.1.1 Shuffled

Because our dataset is very imbalanced, we choose here to only retain the epochs with the best MCC. All metrics for every epochs are available in our replication package.

	Fixes	Inducing
Epoch	37	9
TN	866	2638
FP	466	536
FN	52	48
TP	108	53
Accuracy	0.65	0.82
Precision	0.19	0.09
Recall	0.68	0.52
F-measure	0.29	0.15
Precision-Recall AUC	0.16	0.06
AUC	0.66	0.68
MCC	**0.21**	**0.16**
Loss	0.01	1.75
Val loss	0.01	1.74

We can see that the two models have a low MCC score and thus have rather bad performances when testing them on each other datasets. The most noticeable differences between the two models are the Loss and Validation Loss. Indeed, the model trained on fixes has low Loss and Validation Loss, meaning it did not make many bad predictions during the training. On the other hand, the model trained on vulnerability inducing codes had a high Loss and Validation Loss, meaning it was not able to make good predictions even on its own training and validation sets.

5.1.2 10-Fold

Here, we only retain the epochs with the best MCC when testing on the opposite dataset. All metrics for every epoch for both training are available in our replication package.

Table 5.1: 10-Fold models trained on fixes and tested on vulnerability inducing

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Epoch	7	1	1	1	21	1	1	1	1	1
TN	1262	1331	1331	1332	1135	1332	1332	1332	1332	1332
FP	70	1	1	0	197	0	0	0	0	0
FN	110	147	148	148	83	148	148	148	148	146
TP	50	13	12	12	77	12	12	12	12	14
Accuracy	0.879	0.901	0.9	0.901	0.812	0.901	0.901	0.901	0.901	0.902
Precision	0.417	0.929	0.923	1	0.281	1	1	1	1	1
Recall	0.313	0.081	0.75	0.75	0.481	0.75	0.75	0.75	0.75	0.088
F-measure	0.357	0.149	0.139	0.14	0.355	0.14	0.14	0.14	0.14	0.161
Precision-Recall AUC	0.204	0.174	0.168	0.174	0.191	0.174	0.174	0.174	0.174	0.185
AUC	0.63	0.54	0.537	0.538	0.667	0.538	0.538	0.538	0.538	0.544
MCC	0.296	0.258	0.247	0.26	0.266	0.26	0.26	0.26	0.26	0.281
Loss	1.687	2.465	2.469	2.486	1.557	2.484	2.485	2.475	2.465	2.45
Sparse categorical accuracy	0.531	0.448	0.446	0.445	0.556	0.444	0.444	0.446	0.449	0.451
Val loss	1.981	2.005	1.961	1.836	1.552	1.83	1.843	1.889	1.983	2.127
Val sparse categorical accuracy	0.474	0.5	0.504	0.53	0.563	0.524	0.523	0.508	0.484	0.455

We can see we have the best MCC in the #**1** model, with a score of 0.296. Also, both #**1** and #**5** have a lot of False Positives along with their higher amount of True Positives compared to other models, despite being trained on more epochs. The MCC seems to decrease when the model is trained on more epochs, this can be due to a lack of generalization or to differences in patterns of vulnerability inducing commits.

Table 5.2: 10-Fold models trained on vulnerability inducing and tested on fixes

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Epoch	1	2	2	2	5	8	2	1	2	9
TN	3074	2939	3024	3099	2900	2254	3096	3072	3032	2714
FP	100	235	150	75	274	920	78	102	142	460
FN	83	70	75	83	62	29	80	80	68	45
TP	18	31	26	18	39	72	21	21	33	56
Accuracy	0.944	0.907	0.931	0.952	0.897	0.71	0.952	0.944	0.936	0.846
Precision	0.153	0.117	0.148	0.194	0.125	0.073	0.212	0.171	0.189	0.109
Recall	0.178	0.307	0.257	0.178	0.386	0.713	0.208	0.208	0.327	0.554
F-measure	0.164	0.169	0.188	0.186	0.188	0.132	0.21	0.188	0.239	0.182
Precision-Recall AUC	0.053	0.057	0.061	0.06	0.067	0.061	0.069	0.06	0.082	0.074
AUC	0.573	0.616	0.605	0.577	0.65	0.712	0.592	0.588	0.641	0.705
MCC	0.136	0.147	0.161	0.161	0.176	0.159	0.185	0.16	0.217	0.194
Loss	2.998	2.058	2.059	2.06	1.851	1.793	2.056	3.034	2.06	1.75
Sparse categorical accuracy	0.401	0.488	0.488	0.487	0.505	0.511	0.488	0.394	0.488	0.518
Val loss	2.3	1.93	1.903	1.931	1.725	1.635	1.913	2.03	1.945	1.913
Val sparse categorical accuracy	0.456	0.491	0.492	0.5	0.513	0.542	0.493	0.499	0.495	0.491

We can see we have the best MCC in the #**9** model, with a score of 0.217. Also they all have a lot more False Positives than True Positives.

5.2 Medium dataset

The model was trained on the medium dataset with the following settings.

Type	Training pairs (90%)	Validation pairs (10%)	Vulnerable
Inducing	11766	1307	756

Vocabulary size	Sequence length
830	100

5.2.1 Metrics

Here we only retain the epochs with the best and worst MCC. All metrics for every epochs are available in our replication package.

	Inducing Best	Inducing Worst
Epoch	1	27
TN	3254	3192
FP	30	92
FN	185	200
TP	17	2
Accuracy	0.94	0.92
Precision	0.36	0.02
Recall	0.08	0.01
F-measure	0.14	0.01
Precision-Recall AUC	0.08	0.06
AUC	0.54	0.49
MCC	0.15	−0.03
Loss	1.38	$5.67 \cdot 10^{-05}$
Val loss	1.20	$3.51 \cdot 10^{-04}$

Here we can see again a low MCC for the best one at the 1st epoch. After 27 epochs, the model has learned a lot more from its dataset but its MCC on the opposite one has become even worse than for a random algorithm.

5.3 Large dataset

The model would have been trained on the large dataset with the following settings.

Type	Training pairs	Validation pairs	Vulnerable
Inducing	94130	10458	6051

Vocabulary size	Sequence length
1166	100

As we could not train a model on this dataset, there are no results to show.

Chapter 6

Discussion

6.1 Small dataset

6.1.1 Shuffled

RQ1.Are patterns learned by both models similar?

At first, we thought we could not tell much from the **shuffled** models and models trained on this dataset as we cannot really expect a training to be good for this size of dataset. Both models have a lot of false positives and the overall results are roughly the same as random considering the low MCC values.

Actually, the MCC score is calculated on the opposite dataset while the similar Loss and Validation Loss indicate us that the model does not seem to overfit on its own dataset. The low Loss on the model trained on fixes indicate us that it does a good job at predicting functions coming from the Pre-Fix/Fix dataset but, on the other hand, its MCC score on its opposite dataset means it does not properly predict patterns from the Pre-vulnerability/Vulnerability inducing dataset.

Considering these metrics, it seems the models **do not learn similar enough patterns** to be able to make good prediction on their opposite dataset, whether they are good or not at making predictions on their own data.

RQ2.Are vulnerability inducing code suitable for training?

We cannot really conclude something from this dataset as it is too small to expect a good generalization. Also, the model trained on vulnerability inducing code had a high loss regardless of how many epochs it was trained on, meaning it was not able to predict on its own dataset properly. We need to see if predictions are better once trained on a bigger dataset or not. However, if we can train a model that does a good job at predicting vulnerability inducing code, then we could consider vulnerability inducing code depending on **RQ1** results.

6.1.2 10-Fold

RQ1.Are patterns learned by both models similar?

In the **10-Fold** models, the MCC score is still low for every model and the Loss on their own dataset is rather high. However, we still can notice a few things compared to the previous section for shuffled models.

Indeed, we can notice that almost all models get their best MCC score on their opposite dataset in the very few firsts epochs. At this point, their Loss score on their own dataset is at its highest and they still have a lot to learn from it. Considering the low MCC scores, that

probably means the predictions are just random, but it also means that the more a model learn patterns from its own dataset, the worst are its predictions on its opposite dataset.

As for the shuffled models, we tend to consider the patterns learned are different. To the point where learning on one dataset seems to worsen predictions on the other.

RQ2.Are vulnerability inducing code suitable for training?

Once again, we cannot really conclude anything since every model of both categories are not even good on their own dataset. Still we can notice the vulnerability inducing models seem to predict a lot more False Positives on fixes than the other models on vulnerability inducing codes.

A possible reason for this difference would be that a lot of functions modified between the Pre-Vulnerability inducing commits and the inducing ones were not actually vulnerable. The model would then consider some functions from the fixes as vulnerable because the same functions were present in our vulnerability inducing dataset and labelled as vulnerable.

That would mean our dataset of vulnerability inducing code could contain a lot of **noise** and not be suitable for training. However, it would not mean vulnerability inducing code are not suitable for training but only that our approach for the labelling need to be improved.

6.2 Medium dataset

RQ1.Are patterns learned by both models similar?

Once again, our vulnerability inducing model is unable to predict vulnerabilities from the fixes dataset even with more training data.

The dataset used for training is almost 10 times bigger than our small dataset, however the results are similar. The model has its best prediction of the opposite dataset at the first epoch and this prediction is almost the same as random. We can also notice that the worst MCC score of the model were after 27 epochs and indicates a performance even worse than random despite the very low Loss and Validation Loss indicate a rather good performance on its own dataset.

It is important to keep in mind that the medium dataset is still small to have a good training. Though, this dataset is also a lot bigger than our small one and still shows the same kind of results, making us more confidant that the patterns learned are not similar.

RQ2.Are vulnerability inducing codes suitable for training?

This time our Loss and Validation Loss scores are clearly decreasing during the training. With the medium dataset, the vulnerability inducing model seems to really learn from it without overfitting, as both Loss scores face roughly the same decrease.

With a big enough dataset, vulnerability inducing code could be suitable for training.

6.3 Limitations

Some of these results could be affected by the way we label functions in our datasets. Indeed, every function modified or added between two files of the same pair were considered as vulnerable, however this could actually be simple refactoring or addition of new features that do not induce the vulnerability.

In case of the medium dataset, we also have no guarantee of the quality of our dataset as we use an SZZ algorithm, whose accuracy is rather low. That means we could have pairs of

functions that are actually both vulnerable. To overcome this, we could take the unit tests made for the fix and run them on the pre-vulnerability and vulnerability inducing codes. However, in order to do this, two conditions must be met:

- Unit tests have been made to verify the fix.
- The involved functions specifications did not change between the pre-vulnerability inducing version and the fixed version. Otherwise the unit test implementation may not work for all versions.

6.4 Summary

RQ1.Are patterns learned by both models similar?

Our results on the small and medium datasets showed that the different models are unable to properly predict the vulnerabilities from their opposite dataset. This inability to predict patterns of the opposite dataset is even worse when the model is getting better at predicting its own dataset.

Patterns learned by both models are **NOT** similar.

RQ2.Are vulnerability inducing codes suitable for training?

Our results on the medium dataset showed us that a model can be trained on vulnerability inducing codes and being able to learn without overfitting.

Considering patterns learned are also **not** similar between vulnerability inducing codes and fixes, this training could actually be more relevant to predict or inject more realistic vulnerabilities.

Vulnerability inducing codes **are** suitable for training.

Chapter 7

Threats to validity

7.1 External validity

- Our large and medium datasets are only made of code from the Linux-Kernel repository, thus any model trained on them may not generalize well to other projects.

7.2 Internal validity

- **Noise due to uncertainty.** Although the fixing commits and the vulnerability inducing commits from the small dataset have been verified manually, we cannot be sure these are absolutely corrects. A fixing commit could not correctly fix the vulnerability or could induce another one. There could be other commits inducing the vulnerability before the labelled vulnerability inducing commit, or there could be another vulnerability in the previous commit.
- **Noise due to V-SZZ.** The use of the V-SZZ to generate the larger dataset likely create a lot of noise, considering the low accuracy of this algorithm. Some of our vulnerability inducing codes may actually not be vulnerability inducing, meaning we could actually have some pairs where both functions are vulnerable.
- **Noise due to our categorization of functions.** We assume that functions that were modified by the vulnerability inducing commit or from the fixing commit are vulnerable. Though, we cannot be sure of this. The modification could also be due to refactoring or the addition of a new feature that does not induce the vulnerability.
- **Lack of direct association between the two datasets.** The pre-fixing/fixing commits dataset is larger than the pre-vulnerability inducing/vulnerability inducing commits dataset. This size difference exists because of 3 reasons:
  - V-SZZ sometimes could not find any vulnerability inducing commit.
  - Vulnerability inducing commits are sometimes made by the creation of new files and features and so there is not any previous commit for these files.
  - We could not download some commits that could not be found by PyDriller or Git modules for python.
  - In some cases, files were renamed between commits and we could not always find the previous name.
- **Lack of direct association between our large dataset and the TROVON dataset.** VulData7, that was used in TROVON to retrieve vulnerability fixing commits from the National Vulnerability Database (NVD), is not working anymore. We made our own alternative, but there can be differences between the original dataset from TROVON and our large one, that was made to be compared with the TROVON model.

Chapter 8

Conclusion

To conclude, during this thesis we showed that patterns learned from fixes are very different from the ones coming from the code that has introduced the vulnerability and so are more realistic. A model trained on vulnerability inducing codes is unable to predict properly vulnerabilities from a dataset of fixes, as well as a model trained on fixes is unable to predict properly vulnerabilities from a dataset of vulnerability inducing code for the same projects.

We also showed that a model trained on a large enough dataset of vulnerability inducing codes, even with some noises in it, is able to learn from it and predict unknown vulnerability inducing codes. We conclude that machine learning techniques for vulnerability prediction, as well as vulnerability injection, may benefit more from training on vulnerability inducing code than on fixes as it is usually done.

Finally, we have some suggestions for future works to improve the one done in this thesis.

Equalize the fixes and inducing datasets for more relevant comparison. During this thesis, we made the different datasets without taking into account whether or not a pair of pre-fixing/fixing functions always had its corresponding pre-vulnerability/vulnerability inducing functions pair, resulting in a large size difference between datasets. Making the two datasets so that we always have the exact same functions in both, with the only difference being their version. That way we reduce the differences between the two datasets and we only keep the one that interests us, thus the versions of the different functions.

Find a better function categorization. As mentioned in internal threats to validity (7.2), the way we label the functions in our datasets present some flaws. As we only check if a function was modified by the vulnerability inducing commit or from the fixing commit to label it as vulnerable, refactoring and irrelevant additions are also labelled as vulnerable. We need to find a way to be aware of these irrelevant changes and to not label them as vulnerabilities.

Fine-tuning a pre-trained model. We could try again by fine-tuning another, bigger model as done in the VULGEN [15] paper with CodeT5 for their purpose. Since our datasets are too small to expect a good generalization, we could get different results that way.

Bibliography

[1] Lingfeng Bao et al. "V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2352–2364. ISBN: 9781450392211. DOI: 10.1145/3510003.3510113. URL: https://doi.org/10.1145/3510003.3510113.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[3] Markus Borg et al. "SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project". In: *CoRR* abs/1903.01742 (2019). arXiv: 1903.01742. URL: http://arxiv.org/abs/1903.01742.

[4] Eric Brill and Raymond J. Mooney. "An Overview of Empirical Natural Language Processing". In: *AI Magazine* 18.4 (Dec. 1997), p. 13. DOI: 10.1609/aimag.v18i4.1318. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1318.

[5] Mohamed Amine Chéragui. "Theoretical Overview of Machine translation." In: *ICWIT* (2012), pp. 160–169.

[6] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recognition Letters* 27.8 (2006). ROC Analysis in Pattern Recognition, pp. 861–874. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2005.10.010. URL: https://www.sciencedirect.com/science/article/pii/S016786550500303X.

[7] Aayush Garg et al. "Learning from what we know: How to perform vulnerability prediction using noisy historical data". In: *Empir. Softw. Eng.* 27.7 (2022), p. 169. DOI: 10.1007/s10664-022-10197-4. URL: https://doi.org/10.1007/s10664-022-10197-4.

[8] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. ISSN: 0899-7667. DOI: 10.1162/089976600300015015. eprint: https://direct.mit.edu/neco/article-pdf/12/10/2451/814643/089976600300015015.pdf. URL: https://doi.org/10.1162/089976600300015015.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[10] Ahmed Khanfir et al. "IBiR: Bug-Report-Driven Fault Injection". In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Mar. 2023). ISSN: 1049-331X. DOI: 10.1145/3542946. URL: https://doi.org/10.1145/3542946.

[11] JohnR. Koza. "Genetic programming as a means for programming computers by natural selection". In: *Statistics and Computing* 4.2 (1994). DOI: 10.1007/bf00175355.

[12] Jonathan I. Maletic and Michael L. Collard. "Exploration, Analysis, and Manipulation of Source Code Using SrcML". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 951–952.

[13] Makoto Nagao. “A framework of a mechanical translation between Japanese and English by analogy principle”. In: *Artificial and human intelligence* (1984), pp. 351–354.

[14] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. “The impact of refactoring changes on the SZZ algorithm: An empirical study”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 380–390. DOI: 10.1109/SANER.2018.8330225.

[15] Yu Nong. *VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning*. Jan. 2023. DOI: 10.5281/zenodo.7569854. URL: https://doi.org/10.5281/zenodo.7569854.

[16] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm”. In: *Information and Software Technology* 99 (2018), pp. 164–176. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584917304275.

[17] Michele Tufano et al. “An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation”. In: *CoRR* abs/1812.08693 (2018).

[18] Michele Tufano et al. “On Learning Meaningful Code Changes via Neural Machine Translation”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. Montréal, Candada, 2019.

[19] Haifeng Wang et al. “Progress in Machine Translation”. In: *Engineering* 18 (2022), pp. 143–153. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2021.03.023. URL: https://www.sciencedirect.com/science/article/pii/S2095809921002745.

[20] Qi Wang et al. “A comprehensive survey of loss functions in machine learning”. In: *Annals of Data Science* 9.2 (2020), pp. 187–212. DOI: 10.1007/s40745-020-00253-5.

[21] Yue Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: *CoRR* abs/2109.00859 (2021). arXiv: 2109.00859. URL: https://arxiv.org/abs/2109.00859.

[22] Laura Wartschinski et al. “VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python”. In: *Information and Software Technology* 144 (2022), p. 106809. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106809. URL: https://www.sciencedirect.com/science/article/pii/S0950584921002421.