MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE

Automatic vulnerability injection using Natural Language Processing

PETIT, Benjamin

Award date:
2022

Awarding institution:
University of Namur

Link to publication

Université de Namur
Faculty of Computer Science
Academic Year 2021-2022

Automatic vulnerability injection using Natural Language Processing

Benjamin Petit

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur.

Abstract

PETIT, BENJAMIN. Automatic vulnerability injection using Natural Language Processing. (Under the direction of Gilles Perrouin)

Fault injection is a technique used in the field of Software Engineering for various purposes, such as test assessment, fault tolerance assessment, analysers evaluation and data corpus augmentation. The technique consists on introducing changes to the input source-code in order to alter its expected behaviour, thus, generating multiple faulty versions of a program. The main challenge of such techniques is to find where (source-code locations) and what (source-code transformations) to inject to introduce specific faults or flaws in the program. Typically, these changes are induced by applying mutation operators (code transformation patterns), that are usually based on the language syntax. Therefore, using Natural Language Processing (NLP) techniques and more precisely a Machine Translation (MT) one, we propose an approach that captures automatically the vulnerability intent of the code and, based on a dataset of known vulnerabilities, modifies the code to inject these flaws. This work proposes a complete data processing workflow starting from a corpus of vulnerable and benign code to obtain a dataset that can be exploited by machine learning techniques. Moreover, we propose different ways to augment a dataset by making various syntactic changes without altering the semantics. Finally, our results show that our approach can insert up to 99% of vulnerabilities in the same dataset.

L'injection de fautes est une technique utilisée dans le domaine du Génie Logiciel pour divers objectifs, tels que l'évaluation des tests, l'évaluation de la tolérance aux fautes, l'évaluation des analyseurs et l'augmentation de corpus de données. Cette technique consiste à introduire des changements dans le code source d'entrée afin de modifier son comportement attendu, générant ainsi de multiples versions défectueuses d'un programme. Le principal défi de ces techniques est de trouver où (emplacements du code source) et quoi (transformations du code source) injecter afin d'introduire des fautes ou des défauts spécifiques dans le programme. Typiquement, ces changements sont induits par l'application d'opérateurs de mutation (patterns de transformation du code), qui sont généralement basés sur la syntaxe du langage. Par conséquent, en utilisant des techniques de Traitement Automatique du Langage (TAL) et plus précisément de Traduction Automatique (Machine Translation), nous proposons une approche qui capture automatiquement l'intention de vulnérabilité du code et, sur la base d'un ensemble de vulnérabilités connues, modifie le code pour injecter ces failles. Ce travail propose un flux complet de traitement des données à partir d'un corpus de code vulnérable et bénin pour obtenir un ensemble de données qui peut être exploité par des techniques d'apprentissage (Machine Learning). De plus, nous proposons différentes manières d'augmenter un jeu de données en effectuant divers changements syntaxiques sans altérer la sémantique. Enfin, nos résultats montrent que notre approche peut insérer jusqu'à 99% des vulnérabilités dans le même jeu de données.

Acknowledgements

This internship allowed me to discover the world of Software Engineering and to couple it with the finality of my studies, the Data Science. I discovered the field of software testing while expanding my knowledge in deep learning through its various applications in NLP. But in addition to that, this internship gave me possibility to meet and exchange with great people.

First, I would like to thank Professor Gilles Perrouin from the University of Namur for the opportunity he gave me to perform my internship and for the crucial support he gave me throughout these 4 months. I would also like to deeply thank Ezekiel Soremekun, a research scientist from the Serval team who was always present to give me his advice and answered my questions. I would also like to thank Mike Papadakis which helped me to make my work evolve in the right way.

I am also grateful to the members of the Serval team, Guillaume Haben, Ahmed Khanfir and Aayush Garg. They helped me each in their particular domain of expertise and gave me their valuable advice. They did not hesitate to take their time to guide me and this makes me particularly grateful to them.

I would also like to thank the other students with whom we shared a common office and with whom we exchanged in good mood our respective subjects. Our discussions allowed me to obtain precious ideas and advice that helped me greatly in my work.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

In the Software Engineering domain, various techniques exist in order to prevent the software from bugs or vulnerabilities. Indeed, the security of a software is an important concern. A well-known vulnerability example is the Integer Overflow, a simple vulnerability that can be exploited by an attacker to modify the behaviour of a program. Not protecting a software sufficiently can therefore compromise an entire system and have disastrous repercussions on the system and its users.

This is why various techniques are put in place to prevent these vulnerabilities from appearing in software, such as software testing or the use of static analysers. However, these techniques must also be tested to ensure their ability to discover vulnerabilities in various code contexts. It is in this perspective that fault injection techniques are used. The principle is simple: inject vulnerabilities into code to ensure that detection techniques are able to discover them.

However, most fault injection techniques are based on simple faults or bugs and seldom target security vulnerabilities. Moreover, these techniques are often simply based on the syntax of the programs or use symbolic execution to determine where to inject the vulnerabilities [1, 2, 3]. Unfortunately, they rarely try to catch the “semantics” of the vulnerabilities.

Vulnerability semantic definition : In this master thesis, we define the semantics as the way the vulnerability is intended by the attacker, i.e., its context. Our goal is to have an ML algorithm learn this semantics and the resulting model be able to know how to modify a piece of code to insert a vulnerability, as a programmer could do it.

When used alone, “semantics” refers to this definition. In Section 3.5, we will also introduce the notion of semantic distance. In that case, we will always use this substantive to avoid confusion.

1.2 Research objectives

The main goal of this master thesis is to automatically inject vulnerabilities into Java code. This contributes to creating larger vulnerability datasets to stimulate research on security countermeasures such as static analysers or other tools for detecting security issues. This goal decomposes into two objectives: *(O1)* transforming and augmenting existing vulnerabilities datasets to create semantically-equivalent vulnerabilities in different contexts and *(O2)* consider a machine learning technique able to grasp the semantics of vulnerabilities.

1.3 Research Methodology

Regarding O1, a core issue was to find a vulnerabilites dataset supporting Common Weakness Enumerations (CWEs) in Java. Very few of such datasets exist, but after some searches, our choice was to select the Juliet test suite [4]. This dataset is publicly available and contains a reasonable number of data about realistic CWE vulnerabilities.

However, We found most of the code examples quite simplistic and unlikely to help machine learning towards O2. We therefore had to consider improving this test suite. In particular, we considered two techniques: *(i)* obfuscating the dataset to create semantic clones of the original vulnerabilities [5] , and *(ii)* augmenting the code context via grammar-based fuzzing [6]. We used a static analyser, Infer [7], to ensure that vulnerabilities were kept along transformation and augmentation.

Second, to capture the semantics of the code, we focused on Natural Language Processing techniques, based on the assumption that software code is amenable to such techniques. We finally selected, Seq2seq a form of Recurrent Neural Network (RNNs) already used for security applications [8, 9, 10]. We had to master this technique and evaluate it under different parameters.

Third, we assessed our model on our original and modified datasets. In particular, we considered its ability to create new (unseen) code that is also actually vulnerable. We used semantic distances in the Java bytecode and Infer to measure these two abilities, forming the assessment of O2.

1.4 Thesis structure

This thesis is structured as follows: Chapter 2 presents the Background with the different theoretical concepts needed to understand the thesis. Chapter 3 presents the approach, about all the treatment and analysis done on the data, the different analysers with their results on the dataset and the final workflow of the model. Chapter 4, describes the experiments and answer our research questions. Finally, chapter 5 conclude with the perspectives of this work and the contributions.

2 Background

In the software engineering domain, fault injection is the process of intentionally injecting faults, bugs or vulnerabilities in a piece of code. This technique has several purposes as mutation testing, bug finding systems assessment or the vulnerabilities corpora augmentation.

As this thesis is focused on the security level of the programs, we will talk about vulnerabilities when the piece of code inserted results in a security issue.

This thesis focuses on semantic vulnerability injection. Hence, this section has several interests:

- Explain what exists in the state of the art regarding fault injection.
- Describe the reflexive path starting from what exists in the literature to arrive at a solution adapted to the subject.

2.1 Fault injection

Looking at the state of the art about the fault injection, we can discover different goals and different methodologies for this purpose. The main goals of fault injection are described in the following subsections.

2.1.1 Mutation Testing

The mutation testing is a fault-based technique used to evaluate the test suite of a program. The idea is to assess the covering and the efficiency of the tests about some specific types of faults. The principle of this methodology is to mutate the program to inject some (artificial) faults that must be detected by the tests. To do that some mutation operators are selected depending the type of faults that the tests are supposed to cover. If the tests are able to reveal every mutation in the code, this gives a good confidence to the programmer about the quality of his test suite.

What especially interests us is the mutation testing in a semantic-way and security-awareness.

Semantic Mutation Testing John A. Clark et al. describe a semantic way to implement mutation testing [11]. They call it semantic because they mutate the semantics of the language in which the description is written, instead of the syntax of the description. To do that, they try to mutate a code to insert a fault in a way a programmer could do. By example, a semantic mutation operator could change the precision of a floating-point number or inserting a break statement in a switch.

The goal of this kind of mutation testing is to simulate mistakes that are a consequence of a misunderstanding of the semantics of the language used.

Security aware mutation testing Mutation testing focusing on the security domain is not a new field of investigation. We can find mutation operators specifics to this purpose. For example, in C we can find operators focusing on memory faults [12] or, in Java, we can find PiTest [1], a tool using syntactic transformations to seed vulnerabilities into a Java code. In this later, they propose 15 new mutation operators designed to insert security vulnerabilities. The process to create these new operators has been to find the pattern of some vulnerabilities and create their non-flawed version. It is thanks to these latter that it is possible to find where and how to mutate the code.

The goal of this tool is to reveal the opportunities of vulnerabilities injection and then, can be used to evaluate static analysers too once the vulnerabilities have been injected.

2.1.2 Bug finding systems assessment

The main goal of the next tools is to evaluate bug finding tools.

Lava is a tool developed to inject and validate buffer overflow vulnerabilities in C code. To do so, this tool inserts a bug by looking at situations where an input can trigger a read or write overflow [2].

Bug-injector is a tool able to inject bugs by using bug templates and dynamic traces. It executes some tests on the program to get the traces and find a place respecting a bug template condition. When it finds this place, it is able to insert a bug based on this template [3].

Ibir is a tool using bug reports (to get information about the bug and identify places to inject the bug) and (manually-crafted) patterns to inject bugs in code [13].

Apocalypse is a tool using symbolic execution in order to find inputs and traces and inject bugs when some conditions are met. For the moment, the buggy behaviour inserted is an *assert(false)* [14].

2.1.3 Corpora augmentation

In the dataset augmentation, we can find Evilcoder [15], a tool able to detect where to inject vulnerabilities and remove the security mechanism to make the code vulnerable. It uses static analysers to find sensitive sinks matching bug patterns.

As this tool uses static analysis to inject vulnerabilities, its purpose is more to generate new test corpora than to evaluate bug-finding techniques.

2.1.4 Similarities of most of the Fault injection techniques

Of course some of the tools can achieve multiple purposes, for example PiTest can be used for both Mutation Testing, Bug finding systems evaluation and why not Corpora augmentation.

The main similarities between these tools are that they need to find where to inject bugs and which transformations apply to the code to make it vulnerable.

If we keep looking at the state of the art we can hear about the usage of the Natural Language Processing (NLP) and more specifically the Neural Machine Translation (NMT) in the Software Engineering (SE) domain. The usage of this kind of techniques in the fault seeding gets the advantage of using a model that automatically learn where and what to transform in a program to make it vulnerable.

In addition, some papers are talking about the capacity of the NLP techniques to capture the semantics of the programs [16, 17, 18]. That's why we turned to this direction to find a way to insert vulnerabilities in a semantic-way. By semantic-way, we mean that the model must be able to understand the semantics of the vulnerability to know where to apply the modification that makes the code vulnerable.

Jibesh Patra et al. found a way to inject vulnerabilities in a semantic-aware way and they explain that NMT-based approaches could do it in that way too [19]. This reinforced our idea of using these techniques to inject vulnerabilities. This is what we will be describing in the next part of this Background.

2.2 Fault injection with NLP

Before having a look at the state of the art about the Fault injection using NLP techniques it is important to get a better idea of what it is about.

2.2.1 Natural Language Processing

The Natural Language Processing (NLP) is a branch of the Artificial Intelligence concerning the manipulation and the comprehension of the natural language by a computer in order to complete some tasks.

In the 1950s decade the NLP was mainly based on rules crafted by linguists. Over the years, technologies have evolved and new NLP techniques based on statistics and data have appeared. It is now used by example for Text Classification, Text Extraction, Natural Language Generation, Speech Recognition or even Machine Translation. The latter is the function of the NLP that interests us because its principle is to translate texts from one language to another.

2.2.2 Machine Translation

The Machine Translation (MT), process of translating content, can be divided into multiple types. The main three are:

- Rule-based machine translation: is based on linguistic rules and is the first created system. This technique requires a lot of time to be implemented and the quality of the translation is often quite low.
- Statistical machine translation: uses a bilingual text corpora to analyse existing translations. This technique can statistically translate word by word or, in the last systems, sentence by sentence a text in a language to another one.
- Neural machine translation: uses the neural network methods from artificial intelligence to learn the languages. This is the most accurate and fastest translation method.

In our work, we use this type of machine translation technique - Neural Machine Translation - to inject vulnerabilities in code. The Encoder-Decoder architecture can be used to perform the Neural Machine Translation tasks.

2.2.3 Encoder-decoder architecture

The Encoder-Decoder architecture is an architecture using a Recurrent Neural Network (RNN) as Encoder and another one as Decoder.

RNN The Recurrent Neural Network (RNN) is a type of Artifical Neural Network (ANN). It is used with specific inputs like Time series or, more interestingly in our case, Sequential data. That means it can take a complete sentence as input.

The main idea of this type of ANN is to give to the network the ability of "memory". This is important when the output depends on the previous inputs within the sequence. Like when you want to translate a sentence from a language to another, the translation of a word can depend on the previous words. This makes it particularly used for Natural Language Translation and is, for example, used by the application Google Translate.

To summarise the prediction workflow of a RNN, the model takes a sequence of data x as input and, at each timestep t (at each word), produces a prediction $y^{<t>}$ by using the information of the previous inputs (given by the activation $a^{<t-1>}$) and the current input $x^{<t>}$.

In comparison with a usual feedforward Neural Network, the hidden layers of this model share the same parameters. Hence, the weights are the same between each hidden layer. We can see these hidden layers as one unique layer recursively called for every timestep t in the current input x (so the number of words in a sentence).

The model got its "memory" function thanks to the "hidden state" which remembers the information about the previous inputs. It corresponds to the activation of the previous hidden cell. The Figure 1 summarises the typical architecture of a RNN.

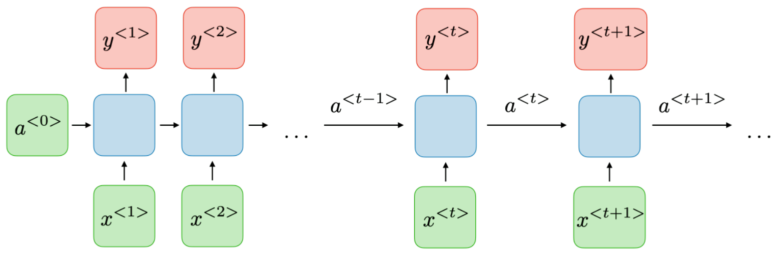


Figure 1: Architecture of a traditionnal RNN
Figure reproduced from [20]

$a^{<t>}$ = activation at timestep t. This represents the hidden state (so the vector containing the information of the previous words). There are different ways to initialize the first values of this initial vector at t=0.

$x^{<t>}$ = input at timestep t (typically a word).

$y^{<t>}$ = output at timestep t (this can be a value corresponding to a word by example in the case of a translation).

blue cell = RNN hidden cell. This cell is composed of a predetermined number of units corresponding to the size of the hidden state (vector). The number of RNN hidden cells will correspond to the number of timesteps. Hence, the number of words in the sequence in input.

Word embedding It is important to notice that the words can't be sent directly as a String to the model. They must firstly go through a word embedding process that converts them in a numerical representation that the model can deal with. There are different methods to create this vector.

One easy to understand method is the One-Hot vector: Each word of the vocabulary is assigned to a bit in a vector. Hence if the vocabulary contains 1000 words, then the vector length will be 1000. Of course with this method, only one bit can be set to 1 in the vector to represent a word.

Then, with the word embedding, the input strings are translated to their representation before being sent to the model and then are translated in the natural language after being predicted by the model.

The main advantages of the RNN

- The memory functionality during the computation that gives to the model better performances when it's necessary to do prediction taking into account the previous inputs.
- The weights share among all the RNN cells.
- The possibility to treat inputs of any length thanks to the recursivity of the Neural Network.

On the other side there are some defects with the memory abilities. This model, for the moment, is not able to remember far previous information and can't take into account the future inputs. Unfortunately it is sometimes useful, in order to predict a word in a sentence (if we keep the example of a text translation), to have in mind the far previous words and the future ones.

Bidirectionnal RNN An improved version of the RNN architecture is the Bidirectionnal Recurrent Neural Network (BRNN). This variant uses a second RNN to compute the input in the reverse order. This allows the initial model to compute predictions with the information of the future context. A quick view of this improved version can be seen on Figure 2

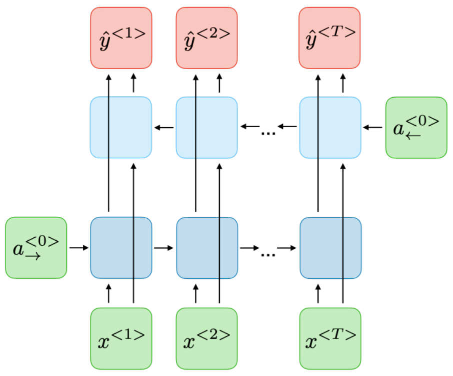


Figure 2: Architecture of a Bidirectionnal RNN
Figure reproduced from [20]

The traditionnal RNN faces the "short-term memory". That means it has some difficulties to take into account the words far from the current state in the sentence, and then uses only the most recent ones to predict. It'll then probably miss important information when trying to predict quite big sentences.

This implies other issues named the "vanishing gradient problem" and the "exploding gradient problem". Indeed, during the training phase, the model updates its weight parameters by computing a gradient thanks to an algorithm. However, the computation of this gradient can sometimes be problematic, making it too close to 0 (vanishing) or too big (exploding). When this happens, the model becomes unable to learn further.

Long-Short Term Memory The Long-Short Term Memory (LSTM) is an improved version of the RNN that has been introduced by Hochreiter & Schmidhuber in 1997 [21]. Its principle is to improve the memory capacity of the RNN. This is done by adding more computation in the RNN cell and by passing more information through the cells. In the RNN, the hidden state is transiting between the hidden cell, but now in the LSTM a new state is transiting: the cell state. We can see the hidden state as the Short Term Memory and the cell sate as the Long Term Memory.

The LSTM cell is divided in 3 gates:

- The forget gate: is used to identify and remove the information from the cell state that are no more useful.
- The input gate: is used to give importance to the new input informations and to choose wich to store in the cell state.
- The output gate: is used to compute the output and the new hidden state.

Hence, the forget and the input gates are used to compute the current cell state corresponding to the Long Term Memory and the output gate is used to compute the new hidden state and the output. These new states are then sent to the next LSTM cell.

To sum up, this new version of the RNN improves the model by making it use more information to predict and thus, avoids the “vanishing gradient problem”.

Gated Recurrent Unit The Gated Recurrent Unit (GRU) is another version of the RNN that has been introduced by Kyunghyun Cho et al in 2014 [22]. It has the same purpose as the LSTM but its architecture is a little different. This cell no longer uses the cell state added by the LSTM but passes all information in the hidden state.

The GRU cell is divided in 2 gates:

- The update gate: does the same job as the forget and input gates of the LSTM. It decides which information to add and which to ignore for the Long Term Memory.
- The reset gate: decides how much the past information must be forgotten for the Short Term Memory.

The choice between LSTM or GRU seems not to be an easy one. Researchers tend to try both and select the one performing the best.

Encoder-Decoder The Encoder-Decoder is a Machine Learning model made of 2 RNN (or alternatives like LSTM, GRU,...). Its architecture is composed of the following components:

- Encoder: a RNN taking one sequence in input and producing a representation (vector) of this sentence. The outputs $y^{<t>}$ of this RNN are not taken into account. The only element we keep from this RNN is the final hidden state corresponding to the Encoder Vector. The number of loops in the RNN will depend on the number of words in the input sentence. (and the number of hidden units in the RNN cells will depend on the desired size of the Encoder Vector).
- Encoder Vector: sometimes named the Context Vector contains the information summary of the input sentence. This is the output of the Encoder and will be the input of the Decoder.
- Decoder: a RNN taking the Encoder Vector as input to produce a new sentence corresponding to the translation of the Encoder input. The number of loops in the RNN will depend on the moment when the prediction will correspond to the end of sentence (e.g.: EOS or <end>).

It is important to precise that the length of the sentences between the encoder input and the decoder input can be different. As the RNN uses input and output vectors with both predefined sizes, this model uses an intermediate vector with a fixed size. Thus, the Encoder uses an unfixed-size vector as input and predicts a fixed size output vector. And this fixed-size vector is then passed as input to the Decoder which predicts in its turn an unfixed-size output vector.

An example [1] of a translation process using an Encoder-Decoder architecture can be seen in the Figure 3. The sentence "The bird fly" is given as input to the Encoder and is sum up into a vector of numbers. Then, this vector is fed as input to the Decoder which outputs its translation "L'oiseau vole".

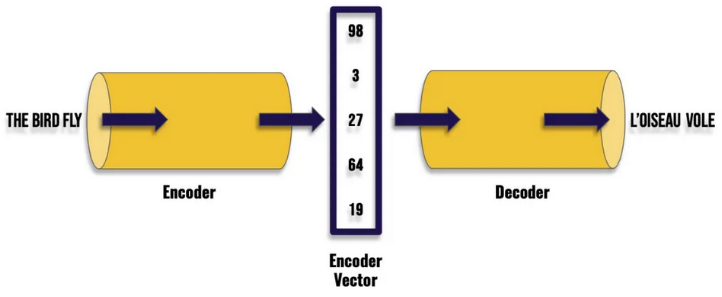


Figure 3: Encoder-Decoder architecture
Figure reproduced from [23]

Sequence to Sequence model The Sequence to Sequence model (Seq2seq) uses the Encoder-Decoder architecture to perform different tasks including the Machine Translation. This model can include some mechanisms in order to raise the performance of a simple Encoder-Decoder architecture.

Copy mechanism The Seq2seq is trained to translate words from a source vocabulary to a target vocabulary (in the case of source code translation those vocabularies can be the same). Then the decoder uses this vocabulary to predict words but this introduces problems for the words out of vocabulary (like the proper nouns). A first solution would be to augment the vocabulary size but this is not a real good solution, especially when working with source code, where the number of different words that can be encountered is much greater than in a natural language. Another, and better, option is the usage of a copy mechanism.

The copy mechanism allows the decoder to select and copy a word present in the input when it fails to predict a word within its vocabulary. This mechanism greatly raises the performances of the model but unfortunately it can happen that even with this mechanism the decoder will not be able to find the word to generate, which will result in a <unk> token instead of a word, in the prediction.

Attention mechanism As explained in the Section about the RNN and the Encoder-Decoder architecture, each time a word is sent to the encoder, it produces a new hidden layer $a^{<t>}$ based on the current input $x^{<t>}$ and the previous hidden layer $a^{<t-1>}$. However, only the last hidden layer, corresponding to the Context Vector, is sent to the decoder. The decoder must then use one single vector to create the sentence to be predicted, regardless of the size of the total input sent to the encoder. The model may then encounter difficulties as the number of words sent to the encoder increases.

[1] `https://inside-machinelearning.com/en/encoder-decoder-what-and-why-simple-explanation/`

The solution could be to increase the size of the vector by increasing the number of hidden units but this will create an overfit on small sentences. That's why the attention mechanism steps in.

The idea of the Attention mechanism is to not only send the last hidden state to the decoder but all of them. Hence the encoder sends all the hidden states to the decoder. To simplify the functioning of the decoder, we can simply say that it pays attention to the most relevant parts of the input sequence at each decoding step.

Beam Search In the traditional Encoder-Decoder architecture, the decoder only predicts the token with the best score. Hence the complete predicted sentence is the sequence of tokens with best scores. But it happens that the word you want it to predict at a step t is not the one with the best score but the second one with a few lower score. That's why we can use the Beam Search. Instead of predicting the token with the best score, it predicts and saves the x best hypotheses. Then, at each new step it looks at the best token that can happen after the x previous hypotheses and keeps only the x new best ones. Thus, once the prediction is complete you can have the x predicted sentences with the best score.

2.2.4 State of the art NMT in Software Engineering

Now that the basis of the NMT, and more especially of the Seq2seq, are in place, we can have a look at what exists in the literature with these techniques in the Software Engineering field. These techniques are used for different tasks in this domain and there are some examples of purposes that we can find.

Program repair The program repair task and debuggings are time-consuming tasks for the software developers. That's why automated program repair tools tend to be created. Some of them are based on the NMT [8, 24, 25] and, trained on bug-fixing corpora, try to fix one line-bugs [8], single statement bugs [25], or even complete methods [24]. The models used on these papers are all based on the Encoder-Decoder architecture.

Vulnerability detection Instead of repairing directly a bug, the Encoder-Decoder architecture can also be used to detect vulnerabilities in the code if there is any [9, 26]. This kind of approach learns and detects the vulnerability context instead of trying to predict a fix. For this purpose the model tries to detect if lines or methods are vulnerable or not.

Some of the other possible tasks Some other tasks are possible in Software Engineering using the Encoder-Decoder architecture. The paper [27] shows some good results with a T5 model (a specific version of the Encoder-Decoder architecture). In these tasks we can find the bug fixing but also assert generation, code summarization and the one that interests us, the vulnerability injection.

In the next Section we will describe more precisely what already exists to inject vulnerabilities with models based on the Encoder-Decoder architecture.

2.2.5 State of the art Fault injection with NLP

CodeBERT [28] is an NLP model designed by Microsoft researchers. This is a pre-trained model for Programming Languages. The usage of this model in the fault injection domain shows its ability to seed "natural" faults that, we can say, semantically resembles to real faults [16]. Effectively, the faults injected resemble to what a real programmer could write (regarding the programmation rules, convention,...) [18].

Codebert can be used for mutation testing [16, 18]. For this purpose they use the Masked Language Modeling task that take a sentence with one masked token and the goal of the model is to find the most likely tokens to replace it. This model can take up to 512 tokens as input and predicts the five best solutions for the masked token as output. The tool being already pre-trained on more than 6 millions programs, it is not especially trained on a dataset containing faults but it proves its ability to create mutants emulating the behaviour of real faults. It can inject faults but not necessarily vulnerabilities (in this work we aim mainly at injecting vulnerabilities).

In the work of M. Tufano et al. [10], they use a bi-directionnal RNN Encoder-Decoder to automatically insert bugs in code. The dataset used to train their model is a corpora of bugs-fixes from GitHub repositories from which they extracted pairs vulnerable-benign methods of maximum 50 tokens. Their fault-injection tool is on the method-level. In fact, one method is supposed to implement only one task and contains enough context for the model. Hence, their model can take as input a method of maximum 50 tokens and should predict the buggy version of this method. They also experiment the same work with larger methods, up to 100 tokens but with lower perfect prediction (when the prediction reintroduces the original bug) (around 6% vs around 20% for the 50 max tokens version of the tool).

Compared to what has been done in the literature, the purpose of this thesis is to inject more specifically security vulnerabilities in code (towards the semantics).

To catch the semantics, the literature shows the potential of NLP models to capture the semantics of a program [16, 17, 18] that's why this is one of these models that will be used in our work.

To focus more on security vulnerabilities than bugs or fault, the choice of the dataset will be crucial. As described in the following Section, the dataset choice and manipulation will be a big part of the work.

3 Approach

This work is based on the SequenceR [8], a tool using a Seq2seq model to repair one-line bugs. However, instead of fixing bugs, the task here is to inject security vulnerabilities. Then our approach is to use a Seq2seq model and assess its capabilities to inject vulnerabilities coming from an appropriate dataset into benign pieces of code.

The main parts of the approach are summarised as shown in the Figure 4 and will progress as follows:

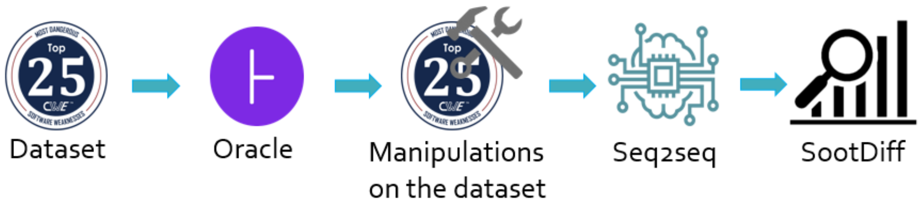


Figure 4: Workflow of the approach

- Dataset: Find a dataset concerning security issues based on the Common Weakness Enumeration (CWE) [29]. This dataset will be analysed, filtered first and prepared to obtain the most suitable instances for the Seq2seq.
- Oracle: The performances of multiple Oracles (tools used to detect vulnerabilities) will be assessed on the dataset. Indeed, it is necessary to have a tool able to classify the instances on which our model is trained. We assume that if the Oracle is able to detect the vulnerabilities in the dataset, it will then be able to detect the vulnerabilities, if there is, in the predictions given by the Seq2seq.
- Manipulations on the dataset: Some manipulations will be made on the dataset to augment it and to evaluate the behaviour of the Seq2seq depending on the modifications applied on the data.
- Seq2seq: The workflow with the Seq2seq will be described. This workflow will describe how the dataset is manipulated with the Seq2seq to finally produce potentially vulnerable data.
- SootDiff: The tool SootDiff will be described to explain how it helped us to analyse the predictions of the Seq2seq.

3.1 Dataset

While reading scientific papers about fault injection, the two most frequently encountered languages are C and Java. For reasons of familiarity with the language and its popularity, we chose to select Java. Once the language was selected, it was then necessary to find a dataset containing as many security vulnerabilities as possible. For each vulnerability, we also needed to have its repaired version.

3.1.1 Dataset description

The selected dataset is the Juliet Test Suite. It has been created by the Center for Assured Software (CAS) of National Security Agency (NSA). Its main purpose is to assess the capabilities of static analysis tools to detect some vulnerabilities coming from the Common Weakness Enumeration (CWE).

This CWE is a collection of known weakness (i.e. security issues) in a software. The Juliet dataset contains 112 of them (for the java version) and more especially 11 of the 2011 CWE/SANS Top 25 most dangerous software errors. The whole dataset represents 28881 "test cases", a name used in this field to describe a piece of code that targets a type of flaw. The test cases have been created either with the "Test case Template Engine" tool or manually on their simplest form from different source files.

The simplest form of one test case of the dataset contains one bad method with the flaw and X non-flawed good methods. As explained in the documentation of the dataset, the test cases are separated in different design (depending, among other things, on whether the vulnerability is in a Java class or not):

- Non-class-based flaw: When the flaw does not concern a class but is defined in methods.
  - with only required methods (NCBFRM): When the flaw and the fix can be contained in one method each.
  - with optional methods (NCBFOM): When the flaw or the fix must use additional methods.
- Class-based flaw (CBF): When the flaw is in one entire class and can't be contained in one method.
- Abstract method flaw (AMF): Use abstract methods (and contains 5 files).
- Bad only flaw: Does not contain the non-flawed version of the flaw.

In its simplest form (NCBFRM), a test case contains one vulnerable method and one or more benign methods with several differences depending on their name:

- good: makes a call to all the other benign methods. Its purpose is simply to run all the different non-vulnerable versions of the test case.
- goodG2Bx: Is the benign version of the vulnerability where the vulnerability is fixed at the source level (when the value of the variable is assigned) but is still potentially vulnerable at the sink level (this term means the moment when the variable is exploited). For example, this may mean that instead of assigning the value of a variable by reading a file, the value of that variable is set manually in the code. If we illustrate this vulnerability correction in the context of a division by zero, it can be by fixing the value of the variable in the denominator statically rather than assigning it dynamically. Thus, the numerator can never be divided by zero during the execution of the program.

Test case design	# files	% of dataset
NCBFRM	14914	32.10%
NCBFOM	27643	59.49%
CBF	31	0.07%
AMF	3473	7.48%
Bad only flaw	113	0.24%
Not defined	289	0.62%
Total	46463	100%

Table 1: Test cases design analysis.
NCBFRM represents the Non-class-based flaw with only required methods. NCBFOM represents the Non-class-based flaw with optional methods. CBF represents the Class-based flaw. AMF represents the Abstract method flaw

- goodB2Gx: Is the benign version of the vulnerability where the vulnerability still has a potentially dangerous source but is fixed at the sink. For example, this can mean adding conditions around an expression in order to check that the value of the sensitive variable meets specific conditions. If we illustrate this vulnerability correction in the context of a division by zero, it can be done by adding an "if" condition verifying that the variable that will be in the denominator in the division is different from zero. Thus, the numerator can never be divided by zero during the execution of the program.

As it is planned to work with one NLP model, it is important to analyse, select and prepare the dataset rigorously.

3.1.2 Dataset analysis & selection

Test cases analysis & selection NLP models require a lot of small data (i.e. with a relatively small number of words) [8, 10]. Then, the choice naturally turned to select the simplest design of the test cases: the "Non-class-based flaw with only required methods".

Thus, this pre-selection of data allows to have, in the dataset, small sentences with directly the vulnerability or the fix in a method. This is important so that the model can have enough context to understand the difference between the vulnerable and non-vulnerable versions of a method while having a relatively small number of words to manipulate.

As the test cases are not ordered according to their design, it was necessary to create a small Python script categorising the Java files w.r.t the documentation.

Table 1 represents the number of files in relation to their design. This Table shows that the NCBFRM test case design represents 32% of the dataset. This is the subset of data selected up to now.

CWE analysis & selection In order to better understand which vulnerability the model is trying to inject, it was decided to keep only a few CWEs . This task should be done carefully as it is important to keep as many files in the dataset as possible to train the model.

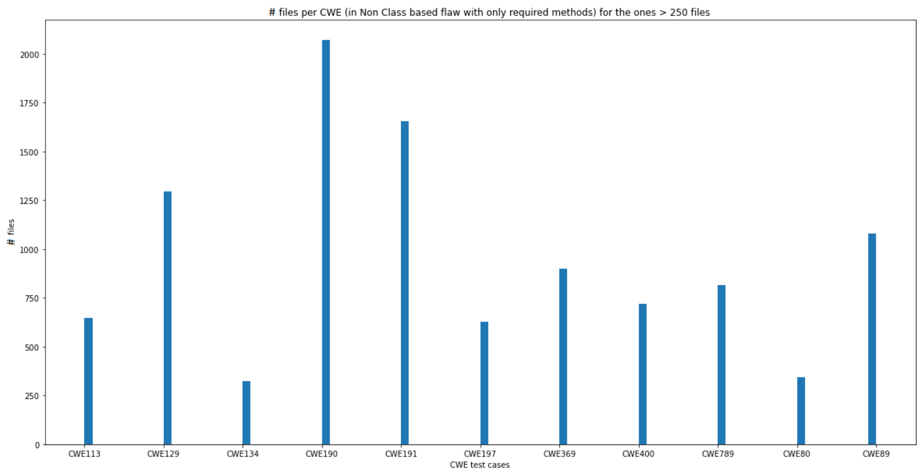


Figure 5: CWE with more than 250 files (NCBFRM)

Then, the idea was to analyse the number of files present in each CWE and to select the most represented ones. This solution allows to have a large number of training instances while reducing the number of vulnerability categories present in the dataset.

The Figure 5 focuses on the CWE containing more than 250 files. The 11 resulting CWE represent an important part of the files (70% of the NCBFRM) and are the ones that will be selected.

Regarding the number of instances in the dataset, it is important to distinguish between the number of Java files present in the Juliet Suite and the number of instances that will be in the dataset. Indeed, the model learns on {benign_method,bad_method} pairs. And a Java file is composed of a vulnerable method with X benign methods corresponding to the non-vulnerable versions of this first method.

As an example, if we look at the file *CWE191 Integer Underflow byte min multiply 01*, it has one vulnerable method and two benign methods (goodG2B and goodB2G). This will give two instances:

$$goodG2B \rightarrow bad$$

$$goodB2G \rightarrow bad$$

Hence for the moment, with these 10477 java files, the dataset gets 33013 instances from the 11 CWE. Every CWE is now shortly described and followed by a short view of the vulnerability and the fix at the sink level if it exists in the dataset (for some of the CWE the only way to fix the issue is at the source-level).

CWE under study

CWE113: HTTP response splitting This CWE is about the possibility of an attacker to control the second HTTP response rendered by the browser by using Carriage Return or (CR) or Line Feed (LF) characters in the data of the HTTP header.

```java
/* POTENTIAL FLAW: Input not verified before inclusion in the cookie */
Cookie cookieSink = new Cookie("lang", data);
response.addCookie(cookieSink);
```

```java
/* FIX: use URLEncoder.encode to hex-encode non-alphanumerics */
Cookie cookieSink = new Cookie("lang", URLEncoder.encode(data, "UTF-8"));
response.addCookie(cookieSink);
```

CWE129: Improper validation of array index This CWE is about the usage of an incorrect array index. An attacker could access to sensitive memory.

```java
/* POTENTIAL FLAW: Verify that data < array.length, but don't verify
    that data > 0, so may be attempting to read out of the array bounds
    */
if (data < array.length)
{
    IO.writeLine(array[data]);
}
```

```java
/* FIX: Fully verify data before reading from array at location data */
if (data >= 0 && data < array.length)
{
    IO.writeLine(array[data]);
}
```

CWE134: Uncontrolled format String This CWE is about the usage of format string as an argument. An attacker could control the format string and this could lead to vulnerabilities.

```java
/* POTENTIAL FLAW: uncontrolled string formatting */
System.out.format(data);
```

```java
/* FIX: explicitly defined string formatting */
System.out.format("%s%n", data);
```

CWE190: Integer overflow This CWE is about a calculation producing overflow. An attacker could exploit this to introduce weaknesses in a resource management or execution control.

```java
/* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
byte result = (byte)(data + 1);
```

```
/* FIX: Add a check to prevent an overflow from occurring */
if (data < Byte.MAX_VALUE)
{
    byte result = (byte)(data + 1);
}
```

CWE191: Integer underflow This CWE is the same idea as the CWE 190 but when the calculation is about subtraction.

```
/* POTENTIAL FLAW: if data == Byte.MIN_VALUE, this will overflow */
byte result = (byte)(data - 1);
```

```
/* FIX: Add a check to prevent an overflow from occurring */
if (data > Byte.MIN_VALUE)
{
    byte result = (byte)(data - 1);
}
```

CWE197: Numeric Truncation error This CWE is about the cast of a primitive into a smaller size. The result of this operation creates a value that can not be trusted by the system.

```
/* POTENTIAL FLAW: Convert data to a byte, possibly causing a truncation
    error */
IO.writeLine((byte)data);
```

There are no B2G for this issue. The only solution, in this dataset, is a G2B and is to manually fix the value of data.

CWE369: Divide by Zero This CWE is about the division by Zero. This can create a crash or a wrong value resulting in a unreliable value for the system.

```
/* POTENTIAL FLAW: Possibly divide by zero */
int result = (int)(100.0 / data);
```

```
/* FIX: Check for value of or near zero before dividing */
if (Math.abs(data) > 0.000001)
{
    int result = (int)(100.0 / data);
}
```

CWE400: Uncontrolled Resource Consumption This CWE is about the improperly allocation of a limited resource.

```
/* POTENTIAL FLAW: For loop using count as the loop variant and no
    validation */
```

```
for (i = 0; i < count; i++)
{
    IO.writeLine("Hello");
}
```

```
/* FIX: Validate count before using it as the for loop variant */
if (count > 0 && count <= 20)
{
    for (i = 0; i < count; i++)
    {
        IO.writeLine("Hello");
    }
}
```

CWE789: Uncontrolled mem alloc This CWE concerns the free allocation of any amount of memory.

```
/* POTENTIAL FLAW: Create an ArrayList using data as the initial size.
    data may be very large, creating memory issues */
ArrayList intArrayList = new ArrayList(data);
```

There are no B2G for this issue. The only solution, in this dataset, is a G2B and is to manually fix the value of data.

CWE80: XSS (Improper Neutralization of Script-Related HTML Tags in a Web Page) This CWE is about the uncorrect checking of the special characters coming from an upstream component. They can then be interpreted as web-scripting elements and be executed client-side.

```
 /* POTENTIAL FLAW: Display of data in web page after using replaceAll()
      to remove script tags, which will still allow XSS with strings
      like <scr<script>ipt> (CWE 182: Collapse of Data into Unsafe
      Value) */
response.getWriter().println("<br>bad(): data = " +
    data.replaceAll("(<script>)", ""));
```

There are no B2G for this issue. The only solution, in this dataset, is a G2B and is to manually fix the value of data.

CWE89: SQL Injection This CWE is about the incorrect checking of special elements coming from an upstream component. An attacker could then modify a SQL request and then alter the database.

```
/* POTENTIAL FLAW: data concatenated into SQL statement used in
    execute(), which could result in SQL Injection */
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
Boolean result = sqlStatement.execute("insert into users (status) values
    ('updated') where name='"+data+"'");
```

```
/* FIX: Use prepared statement and execute (properly) */
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.prepareStatement("insert into users (status)
    values ('updated') where name=?");
sqlStatement.setString(1, data);
Boolean result = sqlStatement.execute();
```

3.1.3 Dataset preparation

Since the whole data preparation process wanted to be automated, the first thing that was done was to check if it was possible to perform each task via a Python script. This approach had the advantage that Python is a high-level language, with a large number of libraries to perform all sorts of tasks and which is widely used in the data world. Hence, if each preparation step can be done with a Python script, they can all be put together in a program that handles the whole data preparation. It is with this in mind that all the following tasks have been designed.

As we can see on Figure 6, the intermediate dataset (The Juliet dataset filtered at the test case level) is going through multiple Python scripts in order to be again filtered (at the method-level) and prepared before being sent in the Seq2seq.

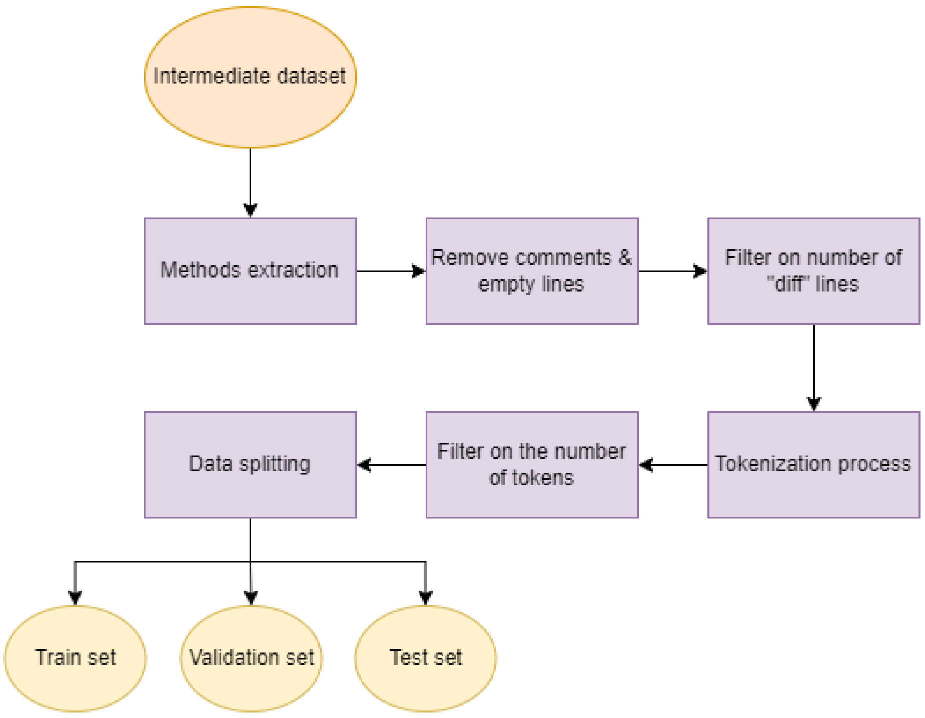


Figure 6: Dataset preparation

Methods extraction The first thing to do when creating the dataset is to separate each Java file into its respective vulnerable and benign methods. Having read in the documentation that (at least in the chosen design) methods never return a value, this means that their signature will always be *void* $< method\_name >$. This information let us know that it would be possible to retrieve the different methods using regex. A small Python script was then made by searching and separating all the methods into text files with the method name in a folder with the test case name. An example of the results of this script can be seen on Figure 7.

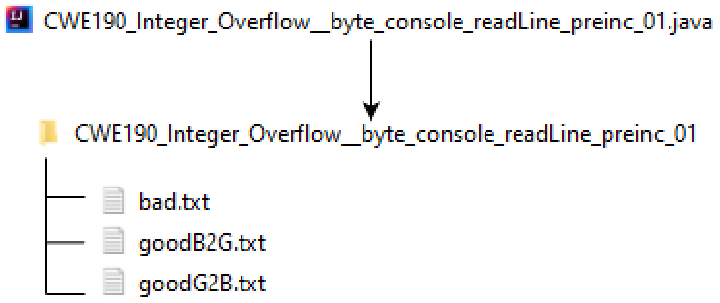

CWE190_Integer_Overflow__byte_console_readLine_preinc_01.java
CWE190_Integer_Overflow__byte_console_readLine_preinc_01
bad.txt
goodB2G.txt
goodG2B.txt

Figure 7: Example of one splitted file

Select the least different methods It was discussed with the team that it would be interesting to focus on methods where the number of lines of code between the vulnerable version and the fixed version is relatively limited. This is to facilitate the training of the model by making it modify only a few lines (or words). To do this, it was thought to use the *diff* command of Linux. Indeed, this command allows to compare two files by analysing them line by line and by proposing a set of instructions (addition, deletion or modification of lines) which allow to pass from one file to the other. By using *diff* in its simplest form, i.e. by simply counting the number of different lines, it was then possible to filter the dataset instances according to "how easy" it would be for the model to switch from a vulnerable method to its benign version. However, before doing so, there was a first preparation work to do. That is to remove the comments and the empty lines.

Remove comments and empty lines A relatively quick way to do this was, as desired, to use Python. Indeed, Python has a built-in method (open) to read files, so we only had to go through the content of all methods and remove the parts we were not interested in. Concerning the comments, this was done via a regex. This solution also has the advantage of being relatively simple and efficient without the need for an additional program. This task can then be perfectly integrated into the process of automatic data preparation via a Python script. Concerning the empty lines, this was done by creating a list of all the lines present in the file being read (by separating the file as soon as a '\n' is encountered). Hence, if the line is empty, it means that it can be deleted.

Apply diff between all benign and vulnerable methods Once the first treatment on the data is done, it is now possible to calculate the number of lines of differences between the methods. This is, again, done with a Python script that runs the diff command on the desired files and saves the output of the command in a file. This allows to create the Figure 8 and to keep the files whose result of the diff command is less than or equal to a chosen limit. (Note that this means that the script must be run on a Linux machine).

As can be seen in the Figure 8, the majority of the benign methods have less than 20 lines of difference with their associated vulnerable method. This is interesting news because it means that there are not too many changes to be made to switch from one method to another. However this is still a significant number. In the experiments we will first try 10 and then 6 lines of difference.

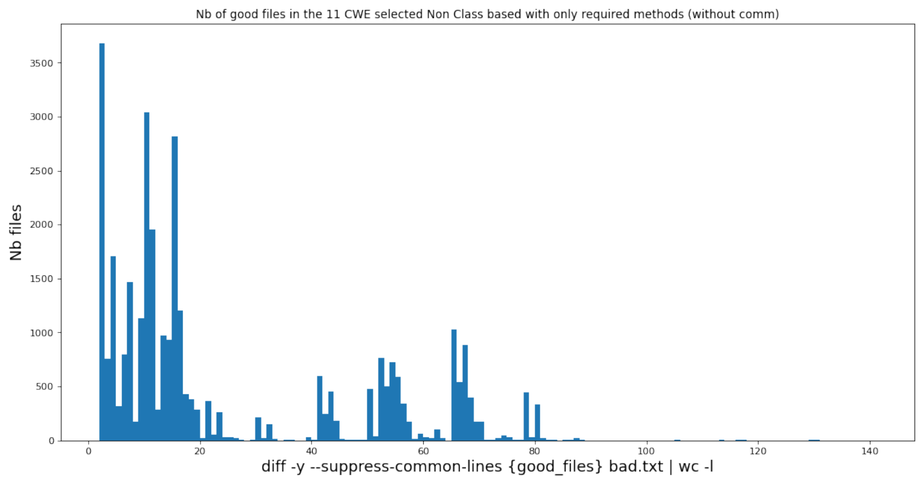


Figure 8: Number of different lines between all the methods

Tokenisation process Once the filtering on the number of different lines is done, the data must be formatted so that they can be used by the model. Indeed, the Seq2seq model used here needs two text files:

- The first one with the sentences of one language.
- The second one with the translation of these same sentences in another language.

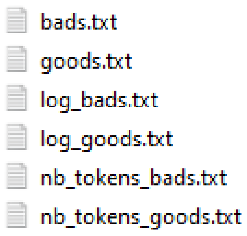
In our case, the first language consists of the healthy methods and the second language consists of the translation of these same methods with a vulnerability (the vulnerable methods). For both files, the tokens must be separated by spaces and the end of the sentence must be terminated by a line break. Unfortunately, in Java the tokens are not always separated by a space. For example, applying a Java method to its object is done with the following expression:

$$object.method$$

It is therefore necessary to separate these tokens into:

$$object\ .\ method$$

That's why the use of a regex was, once again, used to separate these tokens from each other. Hence, passing my data into this new script, we get the different files as seen on Figure 9.



bads.txt
goods.txt
log_bads.txt
log_goods.txt
nb_tokens_bads.txt
nb_tokens_goods.txt

Figure 9: Files after token splitted

- bads and goods are the two files corresponding to my final dataset:
  - goods being all the benign methods written in one line with splitted tokens.
  - bads being all the vulnerable methods written in one line with splitted tokens too.
- log_bads and log_goods are useful to know from which files each method comes from because it will be necessary to reintroduce the model predictions in their original java files.
- nb_tokens_bads and nb_tokens_goods allow to know the number of tokens of each method to get the possibility to filter also on this parameter.

Indeed, previously we talked about filtering on the number of difference lines between the vulnerable and the benign methods, but the possibility to filter on a maximum number of tokens in the methods is also possible and will be used in the experiments described later. Thus, from a folder of test cases, we can filter according to both the number of difference lines and the number of tokens present in the methods.

Split the dataset To avoid the problems of overfitting, underfitting or the biases during the hyperparameter tuning process and the evaluation process, it is necessary to split the dataset. That's why, once the filtering of the Juliet dataset is done, this latter is splitted into subgroups, namely the training set, the validation set and the test set.

This process is done thanks to the Pandas [30] and Sklearn [31] libraries in Python. Indeed, these two libraries are very commonly used in the field of Data Sciences and more particularly in Machine Learning.

Pandas allows the use of a Dataframe, a data structure in the form of a table like an SQL table. This structure allows the insertion of the contents of the vulnerable and benign methods as well as the name of their original file. Once the data is inserted, it is easy to manipulate and can be used for all kinds of analyses, such as the detection of duplicates in a column, which then allowed the deletion of identical methods.

As a reminder, it is important that each benign method is only present once in the final dataset so that the model can train itself on different benign methods. It is the use of this Dataframe that allowed us to notice that some benign methods are identical for different vulnerable methods. This is something we want to avoid in order not to bias the results. Moreover, when increasing the dataset by creating new mutant instances, it is this same functionality that will allow to delete the equivalent mutants, i.e. the programs that are syntactically identical.

Sklearn a.k.a scikit-learn is a Python library providing a range of machine learning algorithms and tools. For this dataset, it was simply used to separate the dataset into several subgroups.

- Training (80%): The part of the dataset that will be used to train the model.
- Validation (10%): The part of the dataset that will be used to evaluate the training of the model and to know if the training allows, on the basis of metrics, to correctly predict unknown instances. It is also this subset that is used to optimise the metaparameters.
- Test (10%): The part of the dataset that will be used to evaluate the model once the training is finished. It is on this part of the data that we will try to compile and verify that the vulnerability has been inserted.

3.1.4 Dataset limitations

As described in the documentation, here are some limitations

The test cases themselves have several limitations

- They are in a very simple form and are relatively straightforward. Despite some code complexity that is inserted in the different versions of a vulnerability, the methods remain very basic. This could have an impact on scanners by "facilitating" the detection of vulnerabilities but it also impacts the Seq2seq model which risks to be able to insert vulnerabilities only in very simple methods too.
- The frequency of vulnerable (and non-vulnerable) methods in the test cases does not represent the reality. Thus, the results of the scanners on this dataset are hardly comparable with the results they could have on real code. Moreover, the results of the Seq2seq to inject vulnerabilities on the test set will not represent the results it could have on real code.

In addition, we noted other limitations

The goodG2B methods As described in the dataset description, there are two main kinds of benign methods. The goodG2B and the goodB2G. As we used the dataset, we realised that goodG2B methods cause several problems:

- First, they strongly change the behaviour of the method. Indeed, instead of dynamically assigning the value of a variable, for example, this value is directly fixed within the method. This is a correction that won't help in a real situation because it changes an important element of the original method.
- Secondly, they do not bring real solutions to the problem. Indeed, if we stay in the example of division by zero, the programmer will probably not want to solve his division by zero problem by fixing the denominator value. The only solution that seems to be interesting for a programmer is to check that the dynamically assigned value is different from zero.
- Thirdly, if we look at the point of view of the Seq2seq, its goal is to inject a vulnerability. If it trains on these files, the only modification it will have to process is to dynamically assign the value of a variable. This does not really make sense in terms of vulnerability injection. This kind of modification could may be called a vulnerability exploitation but this is not the purpose of the model.

This is why these methods will finally be removed from the dataset.

The results of static scanners on this dataset In the next chapter dedicated to Oracles (static scanners) but also in the state of the art, it was shown that static scanners are relatively inefficient to find vulnerabilities in this dataset [32]. Although it was finally possible to use Infer to detect some of them and to establish a small dataset on which to do experiments, this was an obstacle to the progress of the experiments. Indeed, having a tool to detect vulnerabilities is essential to ensure that the Seq2seq model is able to inject vulnerabilities. This is why the size of the dataset that can be used for training, evaluation and testing of the model is directly dependent on the ability of the scanners to discover vulnerabilities in the Juliet Suite.

These limitations led us to do some experiments to improve the dataset and to evaluate the capabilities of Seq2seq on an abstracted (obfuscation) and augmented (Confuzzion) dataset.

3.2 Oracle

The Oracle is the name given to the tool able to detect vulnerabilities in code. The purpose of this section is to describe the different Oracle, or static analysers, that have been tested on the 11 selected CWE.

As explained before, the need of an Oracle able to detect these vulnerabilities come from the need to assess the Seq2seq abilities to inject vulnerabilities. We assumed that if the Oracle is able to detect the vulnerability in the dataset, it will then be able to detect the vulnerability, if there is, in the predicted method given by the model.

During the search of the Oracle, it was not so easy to determine the ability of each of them to detect the vulnerabilities. Indeed, the number of faults detected was sometimes really high and not especially well labelled as what we were looking for. It's then possible that we unintentionally missed some of them.

When speaking about fault detection or more generally classification, some metrics and terminology are used. Let's talk about them.

In the fault detection context, a Positive detection means a vulnerability detected and a Negative detection means no vulnerability detected (then, an healthy code).

True Positive (TP) means that the tool correctly predicted the positive class. In the Fault detection context, this is when the tool detects a vulnerability as a vulnerability.

False Positive (FP) means that the tool incorrectly predicted the positive class. In the Fault detection context, this is when the tool detects a healthy code as a vulnerability.

True Negative (TN) means that the tool correctly predicted the negative class. In the Fault detection context, this is when the tool detects a healthy code as a healthy code.

False Negative (FN) means that the tool incorrectly predicted the negative class. In the Fault detection context, this is when the tool detects a vulnerability code as a healthy code.

From these, we can compute different metrics:

Recall Gives the proportion of real positives that get identified. In the Fault detection context, this gives the proportion of vulnerabilities detected compared to the total number of vulnerabilities in the dataset.

It is computed as follows:

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

Precision Gives the proportion of correct positive identifications. In the Fault detection context, this gives us the proportions of vulnerabilities detected that really are vulnerabilities.

It is computed as follows:

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

3.2.1 SonarQube

SonarQube [2] is a static analyser for 29 different programming languages including Java. It is designed to detect bugs and vulnerabilities and provide the description of the issues.

After some troubles with memory problems and the need of giving him the binaries, it was decided to give it one CWE per one. It also had the advantage of facilitating the search for vulnerability detection on a specific vulnerability.

[2]`https://docs.sonarqube.org/latest/`

Results As we found, this tool was able to detect 5284 CWE89 (SQL injection) and 1968 CWE369 (division by zero) in 3660 and 3050 Java files respectively. While the SQL injection vulnerability was very easy to find, the divide-by-zero vulnerability was more deep in the results and almost went unnoticed.

Our results are in line with the previous work of Alqaradaghi et al. [32]. For their experiment, they try to detect the vulnerabilities of some CWE of the Juliet Test Suite and we have 3 of them in common (the CWE 89, 190 and 400). They detected 5220 vulnerabilities for the CWE89 with this tool, including 2200 TP for 3000 FP. They also explain that this tool does not find any vulnerability for the CWE 190 and 400.

We can then assume that if it does not detect any vulnerability for the CWE 190 about the Integer Overflow, it won't find anything for the CWE 191 about the Integer Underflow.

To sum up, this tool seems able to only find, on the 11 CWE, the SQL Injection (CWE 89) with a lot of FP.

3.2.2 Spotbugs

Spotbugs [3] is a static analyser looking for bugs in Java code. This is an upgraded version of another known tool named FindBugs.

This analyser had the advantage to be launched via a .jar file and comes with a graphical interface that makes it easier and faster to use.

Results As we found, this tool was able to detect 5220 CWE89 and the reference file gets the same result like the previous tool (SonarQube). Of course then the TP and FP rate are the same than the SonarQube.

In addition, we found some CWE129 about the Improper Validation of Array Index. This tool seems able to detect 1988 "Possible null pointer dereferences" and 123 "Array index out of bounds" so 2111 vulnerabilities in 4392 Java files. The TP and FP rate is unknown but looking at the results we can find detection for some of the benign method. Hence the FP rate wouldn't be 0.

To sum up, this tool has, for our current dataset, the same results as SonarQube with the detection of some of the CWE129.

3.2.3 IntelliJ

IntelliJ [4] is an Integrated Development Environment (IDE). That means that its first purpose is to give the ability to a programmer to easily write code in a specific language. This tool has the advantage to own a analyser.

Results As we found, this tool was able to only detect 68 CWE369 corresponding to some of the division by zero.

[3]`https://spotbugs.github.io/`

[4]`https://www.jetbrains.com/idea/`

3.2.4 Hand-made Oracle

From what was seen on the work of Alqaradaghi et al. [32] and what we evaluate up to this moment, we decided to try to create our own hand-made Oracle specific to this dataset.

In order to do that, an analysis of each CWE was necessary. The workflow was as follow.

For every CWE:

1. Find the syntactic difference between the benign and the vulnerable methods (from the point of view of the benign method).
   e.g., in the division by zero context, surround the calculation by a condition verifying the value of the denominator.

2. Find the test that exposes the vulnerability.
   e.g., in the same context, set the value of the denominator variable to 0.

3. Find the behaviour of the code execution that makes me able to detect that the test activates the vulnerability.
   e.g., in the same context, a *java.lang.ArithmeticException* if the result is an Integer.

During the analysis, 6 of the 11 CWE have been decided to be removed. The reason why some are removed will be done during the analysis below.

Analysis of each CWE

CWE80 This CWE contains only goodG2B methods and as explained in the dataset limitations Section, this is not a good point to train the model with these methods.

CWE89

1.
   - Fix the value in the SQL request (G2B)
   - Use *prepareStatement* instead of *createStatement* + setString

2. & 3. SonarQube and Spotbugs can detect them but there is the problem of the FP.

Finally, this CWE won't be kept in the final dataset due to its high number of tokens.

CWE113 This CWE is about Servlet and then focuses on vulnerabilities for Web servers. This is a better idea for the moment to stay focuses on easier vulnerabilities.

CWE129

1.
    - Fix the data value (array index or array size) (B2G)
    - Condition for the array index (>0 and <array.length)
    - Condition for the array size (size of the array before the creation >0)
2. Fix the data value (< 0, > array.length, or 0 depending the case)
3. An exception is raised: *java.lang.ArrayIndexOutOfBoundsException*

CWE134

1.
    - Fix the data value (B2G)
    - Add "%s%n" inside the printf or format method
2. Fix the data value to "Hello World %x"
3. An exception is thrown: *java.util.MissingFormatArgumentException*

With this, the benign method prints "Hello World %x" and the vulnerable method raises an exception.

CWE190 & CWE191 The global idea for both Integer Overflow and Underflow is the same

1.
    - Fix the data value (B2G)
    - Add conditions to assess that the operation with data won't overflow or underflow. But some benign versions are looking for the overflow but not the underflow and vice versa.
2. Fix the data value to < *type_of_data* >. MAX_VALUE or MIN_VALUE depending on the vulnerability we want to inject.
3. Don't send any error.
    - If overflow: Goes to the minimum value of the type(e.g: Byte.MAX_VALUE + 1 = -128)
    - If underflow: Goes to the maximum value of the type.

Thus, the solution has been to do the same operation with the Java.math.BigInteger, a Java object able to store numbers bigger than the limits of the primitive data types. Then it is possible to compare the 2 results and if they are not equal that means that an overflow or underflow occurred. For this purpose, it was necessary to make the hand-made oracle able to detect the operation, translate it into the corresponding BigInteger method and add the correct variables at the good place in the code.

CWE197 For this CWE all the benign methods are G2B, so the fix of the data value and won't be used in the final dataset.

CWE369

1. - Fix the data value (B2G)
   - Add conditions to assess that data (the divisor) is different of 0.
2. Fix the data value to 0.
3. Depends of the cases
   - If the numerator is a double or float type: This result in a really big or small number. If this is a modulo the result is 0
   - If the numerator is another type (like int): This raises an *java.lang.ArithmeticException* Exception

CWE400 This CWE is hard to test and to detect by the nature of the vulnerability. Then it won't be used in the final dataset.

CWE789 For this CWE most of the benign methods are G2B, so the fix of the data value. Then it won't be used in the final dataset.

Oracle creation Now that the CWE are analysed, it is possible to create a Python script able to manipulate the files, inject the test that raises the vulnerability and check the output to define if yes or no the vulnerability has been inserted.

Following the results of the analysis, only these CWE are selected: 129, 134, 190, 191 and 369.
Some of the tests will be inserted manually during the experience with the oracle because of the poor number of test cases. This is the case for the CWE129, CWE134 and CWE369.
For the CWE190 and 191, for which the test is the add of a BigInteger, the process has been automated via a Python script.

With this hand-made oracle, it is then possible to look at the capabilities of the model to inject vulnerabilities in a test set of the Juliet dataset. But the disadvantage is that it is not fully automatised and is really specific to the Juliet dataset.

3.2.5 Infer

Infer [5] is a static analyser developed by Facebook. It is able to detected potential bugs for Java or C (C++, Objective-C) programs. This tool gave some difficulties to be installed, that's why it was not used before the creation of the hand-made oracle that was created to obtain firsts results. But finally, after giving up the idea of install it inside a Docker environment, it was installed in a Debian Virtual Machine and finally worked.
This tool works by giving in input the java source files and the appropriate command to compile them. Then it goes in the capture phase.

[5]`https://fbinfer.com/`

Capture phase This phase uses the compilation process to translates the files in its own internal intermediate language.

Analysis phase Finally, the analysis phase can be launched and Infer will analyse the functions of the intermediate files and report the bugs. For this latest phase, Infer takes arguments to know which bugs it should look for. After some tests on the 5 last CWE and the reading of the documentation, the appropriate arguments were found and especially the *-F* that performs the "no-filtering". With this latest Infer reports the experimental and blacklisted issue types too. This is this argument that makes it able to discover vulnerabilities like the division by zero that, otherwise, are not reported.

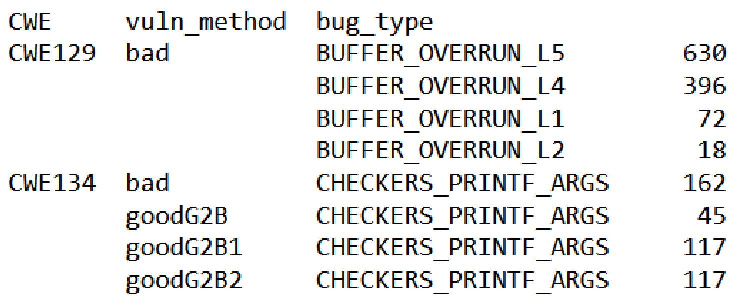
With all these arguments, Infer reports a lot of potential bugs with a specific label corresponding to the type of bugs. The work that has been done was to manually find the types of bugs corresponding to the ones that detect the vulnerabilities we are looking for. For example, the CWE129, corresponding to the vulnerabilities in the array, are reported by the type Buffer_overrun. But there are 6 types of this bug in Infer. Then, after a manual analysis of the reported bugs, it appeared that only the L1,L2,L4 and L5 were due to the real vulnerability of these test cases.

That's why, in order to perform better analysis and save time, the output of the Infer analysis was manipulated with a Python Notebook. This one was created to manipulate the results of the analysis inside a Pandas Dataframe. It was then possible to filter the types of faults that interest us and aggregate the results of the analysis for a better understanding of the results.

An example of the analysis results can be seen on Figure 10 for the CWE 129 and 134.

We can see, in the CWE129 that all the vulnerabilities found are coming from a vulnerable method. That means that we maybe have not detected all the vulnerable methods but that all of the detected vulnerabilities in this CWE are True Positives.

If we look at the CWE134, we can see that Infer detects some (or maybe all) the vulnerabilities in the vulnerable methods but that the goodG2B are often detected as vulnerable too.

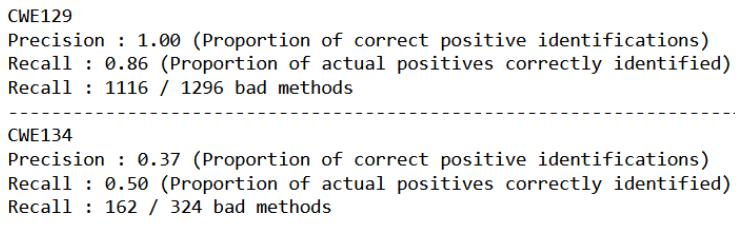


```
CWE     vuln_method  bug_type
CWE129  bad          BUFFER_OVERRUN_L5        630
                     BUFFER_OVERRUN_L4        396
                     BUFFER_OVERRUN_L1         72
                     BUFFER_OVERRUN_L2         18
CWE134  bad          CHECKERS_PRINTF_ARGS     162
        goodG2B      CHECKERS_PRINTF_ARGS      45
        goodG2B1     CHECKERS_PRINTF_ARGS     117
        goodG2B2     CHECKERS_PRINTF_ARGS     117
```

Figure 10: Results of Infer on the CWE 129 and 134

To know the Precision and the Recall of the tool, it is necessary to know the number of each methods present in the dataset sent to Infer. For this purpose, we use another Pandas Dataframe on which we note the presence or not of each method per test case.

Thus, it is possible to compute the precision and the recall of the test cases analysed. An example of the precision and the recall of the analysis can be seen on Figure 11 for the CWE 129 and 134.



```
CWE129
Precision : 1.00 (Proportion of correct positive identifications)
Recall : 0.86 (Proportion of actual positives correctly identified)
Recall : 1116 / 1296 bad methods
--------------------------------------------------------------------
CWE134
Precision : 0.37 (Proportion of correct positive identifications)
Recall : 0.50 (Proportion of actual positives correctly identified)
Recall : 162 / 324 bad methods
```

Figure 11: Recall and precision of Infer on the CWE 129 and 134

As seen before on the Figure 10, the CWE129 has a Precision of 1 because no benign methods get detected as vulnerable, but we can see a Recall of 0.86. That means that not all the vulnerable methods have been detected.

For the CWE134 we can effectively see that the precision is low due to all the benign methods detected as vulnerable and the Recall to 0.50 meaning that the half of the vulnerable methods can be detected by Infer.

Creation of a new dataset for the experiences Now that we found a tool able to discover some of the vulnerabilities, a good way to assess the abilities of the Seq2seq to inject vulnerabilities would be by using the Juliet dataset filtered on the test cases where the vulnerability can be discovered by Infer.

Moreover, the G2B benign methods are too inconvenient to work with, so only the B2G will be selected for the experiences. Hence, the idea is to analyse the whole Juliet dataset and select only the test case with a vulnerability detected by Infer in the vulnerable method and no one in the benign B2G. With the focus on the 5 CWE and after filtering the dataset only on the test case with a detected flaw in the vulnerable method and safe benign methods. This final dataset is made up of 4060 test cases.

This dataset is neither already splitted, nor filtered on the number of different lines and nor on the number of tokens, but this is already a good starting point to evaluate the Seq2seq model more confidentially than with the hand-made oracle.

3.3 Dataset manipulations

It is possible to modify the dataset in order to finally augment it. That's why during the experiments, the behaviour of the Seq2seq model will be evaluated with modifications applied on its data. We tried different dataset alterations. The first one is the abstraction using an obfuscation tool and the second one is the augmentation using a mutation tool.

3.3.1 Dataset Abstraction - Obfuscation tool

In order to abstract the dataset, some tools can do the job and the obfuscator is one of them.

The main goal of the obfuscation is to modify the Java code to make it unreadable and not understandable for a potential hacker. This is done by renaming the name of the variables, the methods and even convert the strings with the ASCII code with some characters. But some other techniques can be used like the insertion of noise code or adding conditions that will never be explored during the execution of the program, but these techniques won't be used here. We do not want to raise the number of tokens of the instances.

In our case, as we only want to add some abstraction to the code, we renamed the variables and methods, and modified the strings by changing some of the characters by their ASCII code. This is done with the tool "Java Obfuscator" from "Semantic Designs" [6].

Problems encountered As we want the obfuscated code to be compilable, it is necessary to have some unobfuscated words. For example, in the case of the CWE134, the methods "printf" or "format" which are the main elements of this CWE mustn't be altered. That's why, a list of reserved words has been provided and adapted to our situation.

As it was not easy to find on the first try all the necessary reserved words, we run the obfuscation tool with a first list, then the test cases were trying to be compiled and all the problematic test cases were manually analysed to find which tokens were missing in the list and caused problems. This was done 4 times before finding every necessary reserved words. After that, the 4060 java files were obfuscated and correctly compilable.

About the ASCII characters, some problems and optimisations were necessary. The compiler sometimes sent errors or did not accept ASCII characters when it expects some specific Strings. That's why some of them were reconverted to their initial value. This was done thanks to a new Python script going through every test case and looking for specific values to replace.

As the graphical interface takes a maximum of 70 java files at a time. This means that if we wanted to obfuscate the 4060 files manually, we would have to execute the programs 58 times. Even more because we had to obfuscated multiple time to find the appropriate reserved words list. Hence we decided to automate the usage of the tool with the command line. Then, this script runs the command to obfuscate one file at a time and we just had to wait the end of the execution.

As the tool changes every word that is not in the reserved words list, the name of the class was modified too. This is problematic because the class name must correspond to the name of the file. Hence, after the execution of the Obfuscator through every test case, the script put back the name of the class in the file.

[6] `http://www.semdesigns.com/products/obfuscators/JavaObfuscator.html`

Result The idea of the usage of this tool was to abstract some words in order that the Seq2seq does not train and predict with always the same variable name. But we needed to assure that the number of tokens was still the same.

After the creation of a Notebook, we could compare the number of tokens in the obfuscated dataset with the number of tokens in the normal dataset. We saw sometimes small differences because the obfuscator changed 0.000001 to $00000010.e - 7$ if we look at the *CWE369 Divide by Zero float zero divide 01* file. This result into 0 . 000001 vs 00000010 . e $-$ 7 once the tokens are splitted. Hence there are 2 more tokens for the same value. It seems to be the only thing conducting to different number of tokens between the 2 datasets.

Finally, here is, an example of a part of code with its obfuscated version of the file *CWE134 Uncontrolled Format String Environment format 01*

```
public void bad() throws Throwable
    {
        String data;
        data = System.getenv("ADD");
        if (data != null)
        {
            System.out.format(data);
        }
    }
```

```
public void bad() throws Throwable
    {
        String I01IOIOIi;
        I01IOIOIi = System.getenv("\101D\104");
        if (I01IOIOIi != null) {
            System.out.format(I01IOIOIi);
        }
    }
```

As we can see on this example, the variable data and the String "ADD" have been obfuscated. These two codes are semantically identical but the obfuscated one is really not easy to read for a human.

A manual verification has been made to verify that ASCII characters are well reconverted to String when going in the compiler. In order to accomplish this verification, the code was compiled and decompiled. The result of this process show us that the Strings are correctly written in the compiled version.

3.3.2 Dataset Augmentation - Confuzzion

To augment the dataset, the Confuzzion tool was used [6]. Basically, Confuzzion is a mutation-based feedback-guided black-box JVM fuzzer focused on the type confusion vulnerabilities.

Let's describe Confuzzion's categorising concepts one by one:

- Mutation-based: Confuzzion applies iteratively different mutation operators to a Java program to create new versions of the same program, also called mutants.

Category	**Mutation operator**	**Description**
Program	AddClassMutation	generates a new Java class.
Class	AddFieldMutation	adds a new field to an existing class.
	AddMethodMutation	adds a new method to an existing class.
Method	AddLocalMutation	adds a new local variable and generates a corresponding object.
	AssignMutation	assigns an existing value to a new variable.
	CallMethodMutation	adds a method call and generates the necessary arguments.

Table 2: Mutation operators of Confuzzion
Table reproduced from [6]

- Fuzzer: Tool that attempts to find bugs and errors in a program by executing it with different inputs, which are in our case different java programs.
- Feedback-guided fuzzer: The fuzzer leverages information gathered from the execution results of previous iterations when generating a new input. We often talk about feedback-loop to describe the guidance of the mutator by the results of the executor.
- Black-box fuzzer: This type of fuzzer means that the fuzzer does not know the source code.
- JVM fuzzer: The fuzzer does not target the detection of flaws in the Java programs but inside the JVM. Hence, it does not test one Java program but the software that runs the program. That's why the inputs are not traditional inputs but Java programs.
- Type confusion: This describes the kind of vulnerabilities that Confuzzion is targeting. Such vulnerability when present in the JVM allows to an object, at runtime, to have a type that is uncompatible with its type at compilation time.

To sum up, Confuzzion is a tool able to detect JVM type confuzzion vulnerabilities using mutation and fuzzing techniques.

In our case, the interesting component of this tool is the mutation. We can use this part of the code to create a mutated and expanded version of the current dataset. That is why a big work has been done to understand the Confuzzion code and adapt it to take one testcase as input and create X (= 50) mutated versions of this program.

Confuzzion can apply different types of mutation. Table 2 illustrates its mutation operators, as listed in the paper [6] .

As we are working on the method level, we naturally want to apply only the method category mutation operators. That's why the Confuzzion tool has been modified to apply only the 3 following operators: AddLocalMutation, AssignMutation, CallMethodMutation. Thus, we can expect our mutants to contain new variables and new method calls.

As the work with Confuzzion progressed, several important points to consider were noted in order to create mutants with the desired behaviour.

The next paragraphs detail these five main points to consider.

Encountered issues

1. The seed in source code By design, Confuzzion applies changes to the input-program in the bytecode level. Therefore, we must compile the seeds – the input test case programs that will be operated by the mutator to create the new programs – prior passing them to the fuzzer. Similarly, the generated mutants of Confuzzion come in bytecode format, and thus, must be uncompiled as we are interested in the source code.

2. Mutate the benign methods The mutation is applied to only one method. In this case, we forced Confuzzion to mutate the vulnerable method. But the associated benign methods must face the same changes: if one line is added somewhere in the vulnerable method, the same equivalent line must be added in the "same" place in the benign methods. Otherwise, the methods wouldn't be equivalent (apart from the fact that one is vulnerable and not the other ones).

3. Mutations at the bytecode level As aforementioned, the mutations are applied directly at the bytecode level, which saves valuable computational costs for the fuzzer as it does not need to be compile the mutants in every iteration. In the other hand, skipping the compilation step bypasses the compiler checking of the generated programs and induces invalid mutants. Indeed, once decompiled, all mutants are not necessarily compilable again.

4. Vulnerability removed by the mutation Going through the multiple versions of the program and add some noise can, unexpectedly, remove the vulnerability. This has been noted with the type of one variable that has changed from Byte to Int in one Integer Overflow test case. This modification directly removes the overflow vulnerability.

5. Mutants equivalents The mutants can be equivalent. By equivalent we mean to be syntactically the same and we do not want the same instances in our dataset.

Let's explain the complete dataset augmentation workflow and how we solve these issues.

Dataset augmentation workflow

Use Confuzzion to create mutants First, we create 50 mutants for each test case.

Then, for each of them in the dataset, as all the test cases are now in a same folder, we must remove the package line. Once this is done, we can compile the seed and send this version as input for Confuzzion. We ask Confuzzion to create 50 mutants and we get the mutants in bytecode.

Next, in order to solve the point 1, it will be necessary to decompile all the mutants to have their corresponding source code. This task has been done thanks to the decompiler FernFlower [7]. We selected this latter because this is the one used by IntelliJ and we already noticed that it is able, most of the time, to decompile the mutants.

At this point we got 50 mutated versions of the seed (with only some mutation operations applied on the vulnerable method) and the seed in a new version (by new, we mean that as it went through a compilation and decompilation process, this version is a bit altered. Some variable names may have changed and some other modifications may have been made during the process).

The same mutations must now be applied on the benign methods.

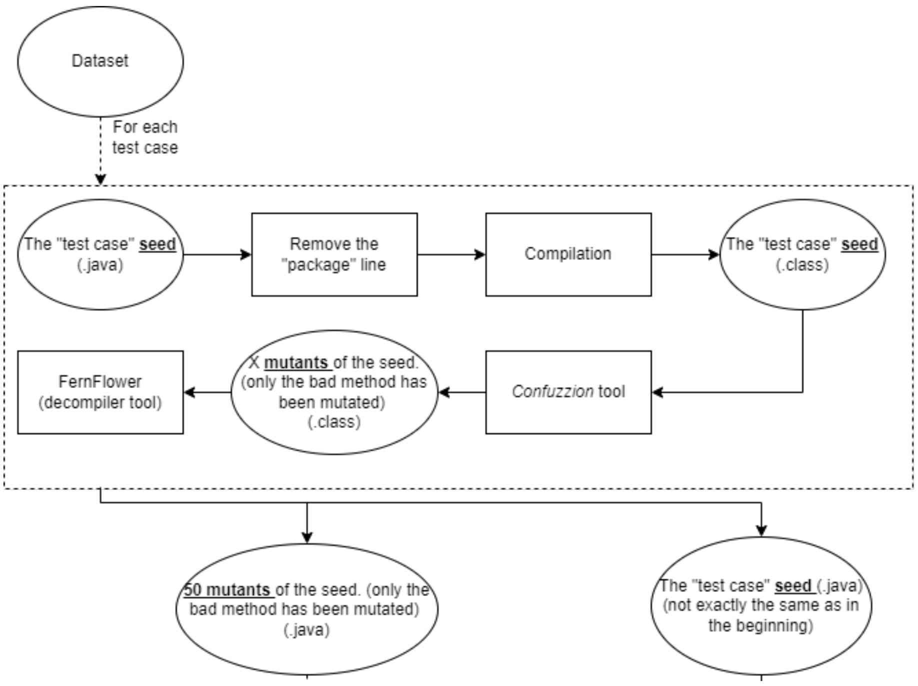

Dataset
For each test case
The "test case" seed (.java)
Remove the "package" line
Compilation
The "test case" seed (.class)
Confuzzion tool
X mutants of the seed. (only the bad method has been mutated) (.class)
FernFlower (decompiler tool)
50 mutants of the seed. (only the bad method has been mutated) (.java)
The "test case" seed (.java) (not exactly the same as in the beginning)

Figure 12: Mutants creation with Confuzzion

Apply the mutations on the benign methods Solve the point 2 was not an easy task.

The idea is to apply the same modifications on the benign methods. To do that, we have to find the operations that have been made between the initial version of the vulnerable method and the version after the mutation.

[7]`https://github.com/sanzibb/fernflower`

To find out the difference, we used again the diff command from Linux. This gives us the operation that have been made between these 2 versions. Gratefully, the patch command exists with Linux and allows to apply a "diff" patch to a file.

Unfortunately we cannot directly apply the patch on the initial benign method. Because the variable names from the version of the seed compiled and decompiled compared to the decompiled version of the mutants are not always the same. Then, we need to find the correspondence between the variable names and set them back. Thus, the diff will only report the operations applied to mutate the vulnerable method and will be applicable on the benign method.

Then we can compute the diff between the vulnerable method of the seed and the vulnerable method of the mutants modified. Thus, we just have to apply the patch on the benign method to reproduce the same mutations.

Once all of this process is done, we have our complete mutants. The last part of the workflow is to remove the unsuitable mutants.

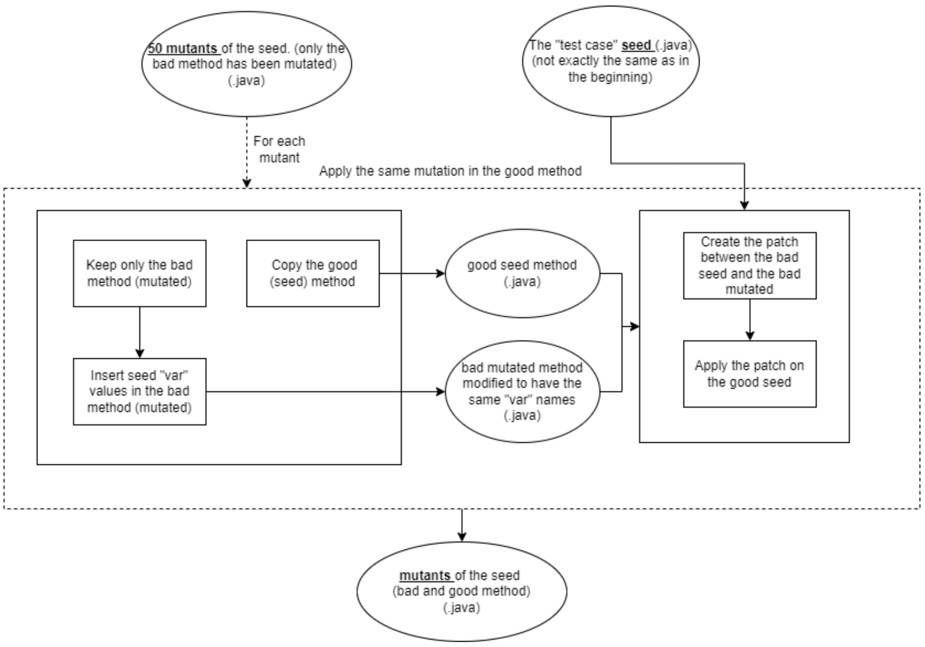

50 mutants of the seed. (only the bad method has been mutated) (.java)
The "test case" seed (.java) (not exactly the same as in the beginning)
For each mutant
Apply the same mutation in the good method
Keep only the bad method (mutated)
Copy the good (seed) method
good seed method (.java)
Create the patch between the bad seed and the bad mutated
Insert seed "var" values in the bad method (mutated)
bad mutated method modified to have the same "var" names (.java)
Apply the patch on the good seed
mutants of the seed (bad and good method) (.java)

Figure 13: Mutation application to the benign methods

Remove the inappropriate mutants At this step, we have 50 mutants of all the seeds (= of all the initial test cases), with the mutations applied on every method. We can see on the Figure 14 how the point 3 and 4 are solved.

In order to solve the point 3, we try to compile every mutant one by one and the ones that get a .class version are the compilable mutants that are going to be selected.

In order to solve the point 4, we use the Oracle over the mutants to make sure that the vulnerability is still in the vulnerable method and that the benign methods are not flawed.

Finally, to solve the point 5, all the equivalent mutants will be removed later during the dataset preparation using the Pandas library [30].

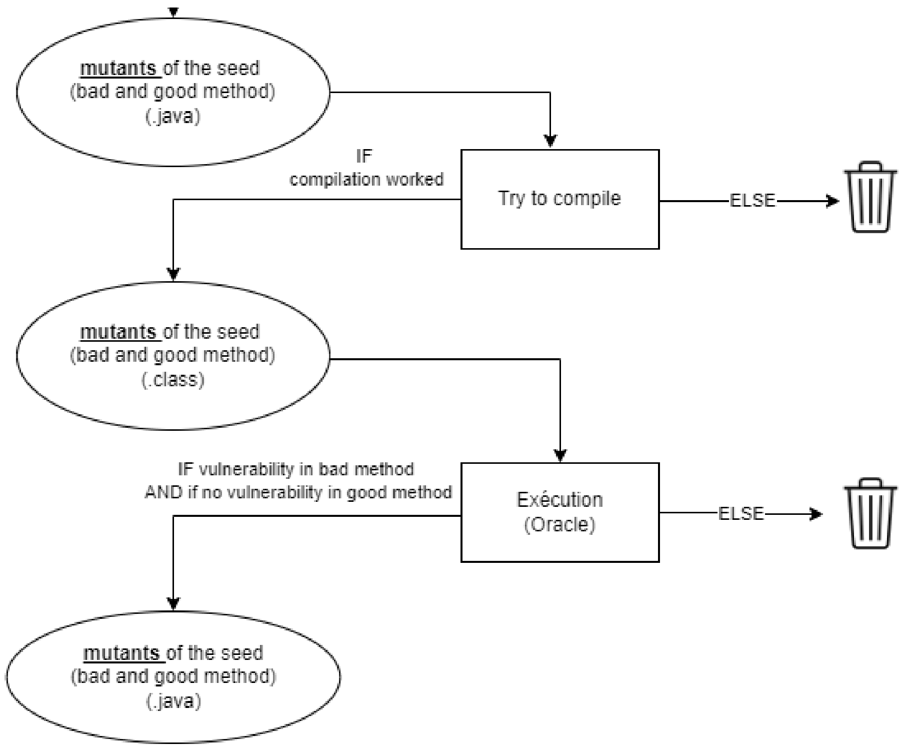


Figure 14: Remove the non-compilables and non-vulnerables mutants

3.4 Seq2seq workflow

Once the dataset is clearly defined and the different sets are created, we can train the model and manipulate its predictions.

3.4.1 Seq2seq training and prediction workflow

The manipulation of the dataset has already been described in the Section 3.1 about the dataset. This process is then summarised in this Workflow.

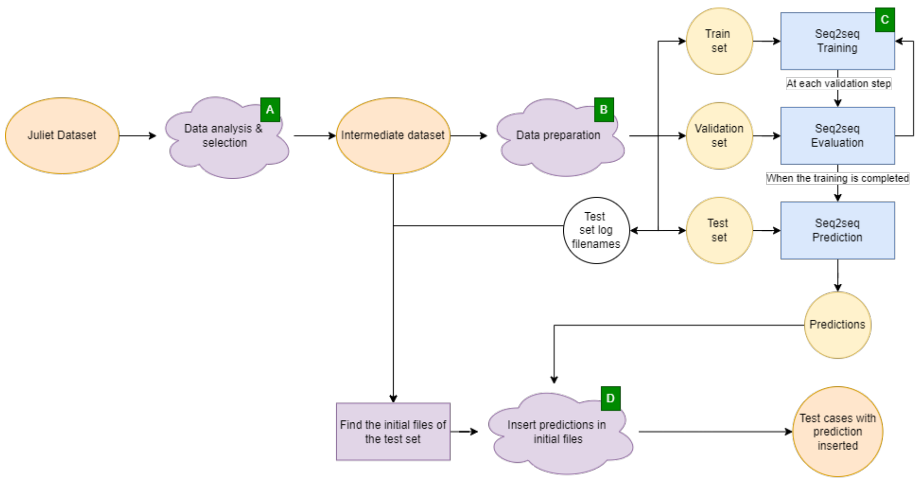


Figure 15: Seq2seq training and prediction workflow
In yellow are the instances at the method-level. In orange are the instances at the "test case" level. In blue correspond the Seq2seq steps. In violet correspond the Python scripts (cloud symbol when multiple consecutive scripts)

In the Figure 15, we can see that the Juliet dataset is going through some process (as explained in Section 3.1) before being splitted in different sets. Once done, the Seq2seq can be trained and evaluated before predicting a potentially vulnerable version of the benign methods coming from the test set. These predictions must then be injected in their initial test case. For that, we can see that a log file has been created linking every method of the test set to its original file. Thus, it is possible to use some scripts to create the test cases with their predicted vulnerable inserted method.

From this workflow, it is important to notice the following points.

Step A: Pre-analysis and selection of the test cases This point sums up the Section 3.1.2 Dataset analysis & selection. A lot of test cases are already removed at this point. Then the intermediate dataset can, for example, contain only 5 CWE because it has been decided to work only with this subset of test cases (this is the case, for example, after analyzing the CWE when creating the hand-made Oracle in Section 3.2.4).

Step B: Data preparation This point sums up the Section 3.1.3.

- The test cases from the dataset are splitted at a method-level (see Figure 7).
- The comments and blank lines are removed.
- The instances are filtered on the number of different lines.
- The tokens are splitted.

- The instances are filtered on the number of tokens.
- The train, validation and test sets are created.

Hence, finally, there are two data selections. A first one is made at a test case level (Point A) when we select the CWE we want to work with. A second one is made at the method level (Point B) depending on the complexity of the methods for the model in terms of number of different lines between the vulnerable and benign versions and the number of tokens.

This number of tokens, calculated when the instances are tokenised, is saved in a file in order to unlock the possibility to analyse the results of the Seq2seq with the information of the number of tokens of the initial vulnerable methods (As the prediction is most of the time an exact matching or close to, using the original vulnerable method was an easy way to understand to capabilities of the model to compile in relation to the number of tokens).

For the last part of the point, we can split the dataset in another way: In order to assess more confidently the Seq2seq model, we can perform a K-Fold Cross Validation.

K-Fold Cross Validation This procedure is used to estimate the skills of a Machine Learning model [33]. It gives the possibility to give a better estimation on how the model is supposed to perform on unseen data. This estimation is usually less biased than a simple train/test split.

The idea of this technique is to divide the dataset into K groups. Then, the model will be trained and validated K times with each time a new validation set (corresponding to the validation fold) and a new train set (corresponding to the K-1 remaining sets). On the Figure 16 we can see an example of a 5 Fold Cross Validation.

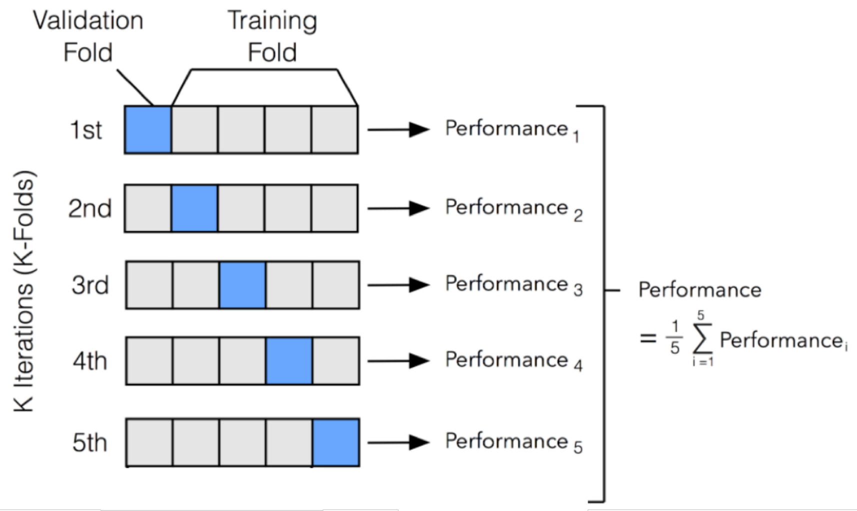


Figure 16: 5-Fold Cross Validation
Figure reproduced from [33]

As we can see, there are no more 3 sets. This new validation set replaces the previous test set, and the previous validation set is included in the training set because it is no more useful: we do not want to assess the performance of the model anymore during the training because the hyperparameters optimisation has already been done.

Step C: Seq2seq Training In order to complete an hyperparameter optimisation, the output of the training is saved inside a text file. This makes us able to gather the metrics at each validation step. Thus, it is possible to plot them on a graph and find the best hyperparameters conducing to better predictions accuracy.

Step D: Insert the predictions inside their initial files Once the benign methods sent to the model are "transformed" by the model into their flawed version, it is necessary to insert them in their initial file in order to execute and analyse them. That's the job done by the point D.

For this, we get the initial test cases thanks to a log file containing the name of these files when the data has been splitted.

Then, the prediction is untokenised. This is necessary to join some token for that the compiler understands the code (e.g. the "++" or "<object.>.<method>").

Once this is done, a script removes the vulnerable method from the initial file (test case) and replaces it by the predicted one. This script checks as well whether these two methods match exactly, by comparing their corresponding tokenised versions side by side.

It is necessary to verify the exact matching when the prediction is still tokenised, as the untokenisation may introduce whitespaces causing the matching to fail and skewing results.

It should also be noted that it is possible for the test set to contain multiple benign methods from the same test case, so a file number indentation mechanism has been added. This allows files to be numbered to distinguish them and avoid conflicts when they are given the same name. Of course this also mean that the class names had to be changed to match the new file names.

Finally, we got all the predictions in their initial test cases and they can be analysed.

3.4.2 Seq2seq prediction analysis methodology

From these predictions, some analyses were necessary in order to check that the whole process has gone well and to understand the results. That's why different Notebooks have been created, the most important ones are described here to give an idea of the methodology.

Juliet predictions analysis This was one of the main Notebooks for the analysis due to its multiple purposes:

- Compute the ratio of identical matching.

- Compute the ratio of compilable files.
- Find the files that compile or not depending on the number of tokens of the original vulnerable method.

Finally this notebook makes us able to verify the results and avoid as much as possible human mistakes resulting in bad interpretations of the results. Indeed, at a moment, some files were not compilable while they were matching exactly the initial method. After a manual inspection, the problem was discovered and came from the untokenisation process that was not complete (some spaces were not removed and then the compiler didn't accept the file). Hence this notebook makes us able to discover and fix this issue.

To also avoid other possible errors, this code make us able to find the compiled files that are different from the original vulnerable version, in order to manually inspect the difference.

Finally, it gives us the filename of the predictions that do not compile while they are less than X tokens. Then a manual inspection can be done to try to understand a possible human reason of this non compilabilty.

Infer results json analysis Once the first notebook was executed without any error or anomaly detected and that Infer was executed on the prediction, this notebook was executed to look at the Infer results.

Its idea is to aggregate the results of Infer into a Dataframe. The output of Infer has the advantage to be readable as a Json, so it is quite easy to manipulate and to create a Dataframe from this one.

In order to have more information, another Dataframe was created gathering the presence of the methods inside each test case. Thus, once the two datasets are merged, it is possible to compute the TP, FP, TN and FN. Then, assuming that Infer detects every vulnerabilities inserted, it is easy to compute the Recall and the Precision of Infer and this gives us the capacity of the model to inject vulnerabilities.

This Notebook was also used to find the files where no vulnerabilities were discovered by Infer in order to do a manual inspection.

SootDiff Another important Notebook was the one about SootDiff, a tool that is explaine in the next Section.

Analysis methodology conclusion To conclude, the results were all computed and analysed using notebooks. It was really convenient due to the nature of the results and the information to gather. But unfortunately a human error could lead to a gap in the results. That's why a big point has been allowed to do manual inspections.

3.5 SootDiff

The usage of the same dataset in different versions (normal, obfuscated and augmented with mutants) in the experiences could conduct to a different behaviour from the Seq2seq and we would like to evaluate this behaviour. What we wanted to evaluate more precisely is the creativity of the model to predict new vulnerable methods according the different manipulations done on the dataset. Hence, for the predicted methods that are not exactly matching the initial vulnerable method, we wanted to evaluate the semantic difference between them. This would make us able to define if alterations on the dataset make the model able to create more innovative predictions.

Then, we wanted to compare the two files at a semantic level. The usual diff command couldn't be used anymore, because the difference would be computed at a syntactic level. In a more semantical way, we thought computing the difference at the bytecode level.

For that we used SootDiff [8]. This tool has been designed to identify if two bytecodes are coming from the same source code. It allows to know if a provided bytecode really matches a source code, in order to be sure that the provided bytecode does not contain vulnerabilities. Because there are no possibilities to validate that a bytecode corresponds to the source code. But the problem by recompiling a source code is that it can produce a different bytecode if it is compiled with a different compiler. Then, SootDiff has been created to assess that two different bytecodes are representing the same behaviour of the program.

This tool shows better results than simple bytecode comparisons thanks to the usage of an intermediate representation (Jimple) and code clone detection techniques.

Soot SootDiff is based on Soot [9]. This later is, initially, a framework that simplifies the task of optimising Java Bytecode. It works by converting a program into an intermediate representation like, in our example, Jimple. Then, this tool can analyse and manipulate this representation and produce different ouputs like Java bytecode.

We can notice than Soot and Jimple are also used by Confuzzion to manipulate and mutate the Java bytecode.

Jimple The principal intermediate representation of Soot is Jimple [10]. it converts the Java code into a 3-address code in a control-flow graph.

SootDiff on the Seq2seq predictions SootDiff works by taking 2 files as input. In our case, the first one is the test case with the vulnerable method replaced by the predicted vulnerable method, and the second one is the initial test case. As these files must be compiled and given to the tool one by one, the process has been automated thanks to a Python script.

This script takes, for every test case, the initial one and the predicted, copy them in the appropriate SootDiff directories, compile them, execute SootDiff, save the output and remove the files. Thus, it's possible to get every semantic difference between the predictions and the initial vulnerable methods.

As example, there is the real vulnerable method of the file *CWE129 Improper Validation of Array Index large fixed array read check min 02.*

[8] `https://dl.acm.org/doi/10.1145/3315568.3329966`

[9] `http://soot-oss.github.io/soot/`

[10] `https://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf`

```
// Real bad method
public void bad() throws Throwable {
        int OO1IOIOIO = 100;
        int[] array = new int[]{0, 1, 2, 3, 4};
        if (OO1IOIOIO >= 0) {
            IO.writeLine(array[OO1IOIOIO]);
        } else {
            IO.writeLine("Array index out of bounds");
        }
    }
```

The following code is the prediction of this vulnerable method by giving to the model one of the associated benign method.

```
// Predicted bad method
public void bad() throws Throwable {
        int OO1IOIOIO = 100;
        int[] array = new int[]{0, 1, 2, 3, 4};
        if (OO1IOIOIO < array.length) {
            IO.writeLine(array[OO1IOIOIO]);
        } else {
            IO.writeLine(" <unk> index out of <unk> ");
        }
    }
```

Hence, if we manually look at the difference, the real vulnerable method is about looking only at the minimum value of the future index but not at the maximum value (index < array.length). And the predicted vulnerable method is about only looking at the maximum value of the future index but not at the minimum (index >= 0). Hence in both cases we could find a vulnerability. But only the real vulnerable method will trigger an exception because the value of this future index is manually set to 100.

The following dictionary is the result when we give to SootDiff the two previous methods in their initial file. “lhs” corresponds to the initial test case and “rhs” corresponds to the test case with the predicted vulnerable method.

```
"lhs": {
"bad.7":[
    "$i1 = $r2[100]",
    "staticinvoke <testcasesupport.IO: void writeLine(int)>($i1)"
]},

"rhs": {
"bad.7":[
    "$i1 = lengthof $r2",
    "if 100 >= $i1 goto staticinvoke <testcasesupport.IO: void
        writeLine(java.lang.String)>("<unk> index out of <unk>")",
    "$i2 = $r2[100]",
    "staticinvoke <testcasesupport.IO: void writeLine(int)>($i2)",
    "goto [?= return]",
    "staticinvoke <testcasesupport.IO: void
```

```
        writeLine(java.lang.String)>("<unk> index out of <unk>")"
]}
```

This example shows that there is a difference between the two files at the instruction 7. We can see that on the real vulnerable method, the code executed will be to print the content of the array at the index 100. If we execute this code, this will produce an exception.

But, for the predicted vulnerable method, the condition is about the maximum limit of the index instead of the minimum. Hence the condition will be evaluated to false and enter inside the else condition that is about printing a warning message.

From these illustrations we can see that the difference is more noticeable when it is done at the bytecode level than if we did it at the source code level.

Note: This predicted vulnerable method will be analysed as non vulnerable by Infer because the value of the index is static and the condition is sufficient to avoid an exception. But if the value of the index is modified by a negative number, the condition won't be enough to avoid the exception. This is this latter point that is missing to have a complete vulnerability inserted.

Compute a semantic distance metric Now that we are able to find the difference between the instructions of the two instances, we would like to compute a semantic distance metric. In order to do this, we thought about computing a metric representing the distance between the prediction and the real code. Then, we decided to compute the Levenshtein distance between all the different instructions.

Levenshtein distance The Levenshtein distance corresponds to the minimum edit distance between two strings [34]. This corresponds to the number of insertions, deletions and substitutions necessary to go from a word to another one.

In our case it is used at the instruction level. Then, for two instructions, it gives us the number of operation to convert the first one into the second one.

Hence, this distance gives us an idea about the distance between two instructions at the bytecode level.

In order to not re-implement the algorithm, we used the one from the site "python-course" [11].

[11] `https://python-course.eu/applications-python/levenshtein-distance.php`

4 Experimentation and Discussion

4.1 Preliminary Experiments

We conducted some preliminary experiments in order to get acquainted with the Seq2Seq model and design our complete experimental protocol.

OpenNMT Seq2seq In order to get started and as our work is based on the Seq2seq used by the Sequencer [8], we started with the Quickstart of OpenNMT to have a first overview of the tool, the format of files and the hyperparameters.

Seq2seq with Juliet dataset (11 CWE) Then, we analysed deeply how the SequenceR works and we prepared an equivalent workflow for our purpose. We started the analysis on the Juliet dataset as explained in the Section 3.1 to finally try the Seq2seq with the 11 CWE. At this moment we saw the limitations of the model on the long sentences.

In order to have a first idea of the capabilities of Seq2seq to inject vulnerabilities, the predictions on the CWE369 about the division by zero were manually analysed. This gave us a good introduction about the capabilities of the model to insert vulnerabilities.

Seq2seq with Juliet dataset (5 CWE) At this step, the model seemed able to inject vulnerabilities as long as it must not predict more than 100 tokens. We then needed to assess more precisely its abilities. As none static analysers seemed to work on this dataset (at this moment), an hand-made oracle was created as explained in the Section 3.2.4. From this, the dataset was again reduced to 5 CWE and a filter of 6 different lines was selected. The results on this were really impressive, our hand-made oracle discovered every compilable prediction as vulnerable. These results were really encouraging but unfortunately using this oracle is not a scalable solution because it is not fully automatised, too specific and there is a risk of errors. That's why we will use Infer as Oracle in the final experiments.

Seq2seq with an hand-made dataset In order to have an idea of the performances of the model on another dataset, 24 methods were hand-written based on the Juliet Test Suite. To do that we took simple methods and added some lines without any impact on the vulnerability, other variable names and different conditions. This gave us a first apriori about the weakness of the model to get out of its comfort zone.

These first experiments make us more confident about both the model and the dataset. It was then necessary to optimise the hyperparameters before going further.

4.2 Seq2seq hyperparameters optimisation

The idea of the hyperparameters optimisation is to find the best hyperparameters that makes the model able to predict the best results for unseen data. As the idea of the model is to finally insert vulnerabilities inside any dataset or piece of code, we decided to evaluate the model by training on the usual test cases and evaluate it on the hand-made dataset.

Dataset used For this purpose we will stay focused on the 5 current CWE, in their obfuscated version in order to complicate the training. Indeed, we assume that train on obfuscated data should help the model to not overfit on the dataset and being more able to inject vulnerabilities into unseen code.

In addition to this, we still select one the goodB2G methods and we filter on a number of lines equal to 10.

Main hyperparameters The main hyperparameters, shared along every experiments, are the following ones:

- Seed: 1. In order to get the possibility to compare the experiments.
- 2 layers in the Encoder and in the Decoder.
- A Bridge layer between the last encoder state and the first decoder state.
- Type of RNN: LSTM.
- Global attention mechanism. In order to improve the abilities of the model.
- Copy Mechanism. In order to improve the abilities of the model (it is probably mandatory to use it, especially when working with obfuscated data).

Metrics The two metrics gathered are the following ones [12][13]:

$$Accuracy = 100 * \frac{self.n\_correct}{self.n\_words}$$

$$Perplexity = math.exp(min(\frac{self.loss}{self.n\_words}, 100))$$

with the loss = the cumulated negative log likelihood of the true target data.

Before starting the tuning Before starting the hyperparameters tuning, we wanted to be sure about some points

Seed As we want to compare the experiments between them, we need to fix a seed [14]. To be sure the seed works as wanted, we launched multiple times the same experiment and this produces the same results. Hence, we conclude the seed works as desired and that we could compare the different experiments between them.

[12]`https://github.com/OpenNMT/OpenNMT-py/blob/7c314f41dc1b017ac105144beeb53cb072960a54/onmt/utils/statistics.py`

[13]`https://opennmt.net/OpenNMT/training/logs/`

[14]We are now talking about the seed in the machine learning field. This corresponds to the initialisation state of the a pseudo-random number.

Validation We wanted to be sure that the selection of the validation set and steps does not impact the training. For the validation set, we run multiple times a same experiment by changing the validation set. The training metrics were still exactly the same so we conclude the choice of the validation set does not impact the training. For the validation steps, we run two times the same experiment but with double the number of steps of the first experiment. The results for the common steps were the same so we conclude the choice of the validation steps does not impact the training.

Then, we could start the hyperparameters tuning.

Hyperparameters to tune

- Earlystopper: We wanted to see if the earlystopper could have no impact on the training but after having tried different parameters, it never stopped the training.
- GRU instead of LSTM: The usage of an GRU seems to have bad results compare to the LSTM (for the Encoder and the Decoder).
- Three layers instead of two: We wanted to define if the usage of more layers makes the model able to predict better results. The training was taking more time but the results didn't get better.
- Vocabulary Size: The vocab size seems to be one of the most important hyperparameters, we tried then different values. The default value is 50.000 but is probably more appropriate for natural language than for code.
  - 100: Bad results.
  - 250: A bit less accuracy but real improvement of the perplexity.
  - 500: A bit worse than 250.
- Batch size: This parameter was fixed to 16 but after having tried 32 and 64 it appears that 32 was the best compromise.
- Word Vector size: This parameter is important because it is the way the words are encoded. This size is 500 by default and was fixed to 256 up to now. After having tried 100 and 512, it seems that keeping 256 is the best solution.
- Remove the bridge layer: In order to have an idea of the impact of this layer, we tried to remove it. The accuracy was a bit worse but the perplexity really better. But after an analyse of the predictions it appeared that the results were horrible. Hence it reinforced the belief that we must not only focus on the metrics but on the predictions too.
- RNN size: The RNN size corresponds to the size of the hidden states and to the length of the Context Vector. This is then the size of the vector containing the information between the Encoder and the Decoder. Its size is by default in OpenNMT (the current library used) 500 and we fixed it to 256 as in the SequenceR paper. We then try 100 and 500 and finally it appears that the results are horrible for 500 but interesting for 100.

In order to remove the bias of assessing the individual impact of each parameter on the performances of the model, we decided to select the most important ones and evaluate the model when modifying simultaneously some of them. Hence we selected those that seemed to be the most relevant.

For these experiments we tried to combine in different ways the values of the batch size, the word vector size and the RNN size.

Finally the most interesting experiment seems to be the one with the following hyperparameters: batch size 32, word vector size 256, RNN size 100 or 256.

Conclusion To conclude the results of the hyperparameter optimisation, the early stopper seems not to have any impact on the training. Use GRU instead of LSTM cells shows bad results on this experiment. Add an additional layer on the encoder and decoder seems not usefull to have better results. Find the appropriate vocabulary size is not so easy and 250 seems to be the good compromise. A bigger batch size (32 or 64) seems to help the model to get better predictions. It seems great to use a Vector size of 256. Finally, the model seems to get better results with a lower RNN size like 100.

The model is able to quickly have good results, but seems to need to train longer to have more accurate results.

It is important to notice that using only the Accuracy and the Perplexity to evaluate the model is not enough. A manual analysis of the prediction is necessary to understand how the model reacts to every modification and this makes the choice of the hyperparameters less precise for us.

4.3 Research questions

Now that the best hyperparameters are selected, it is possible to run experiments in order to answer the Research Questions.

RQ1. How effective is Seq2seq to inject vulnerabilities ? This question is about assessing the general performances of Seq2seq model to inject vulnerabilities. This is done by rating the percentage of compilable predictions, the percentage of exact matching between predictions and the initial vulnerable methods and the percentage of vulnerabilities inserted.

RQ2. Is Seq2seq sensitive to the usage of the same variable names during the training ? This question is about assessing, in a way, if the model is able to catch the semantics of the vulnerability. Indeed, if the model is only sensitive to the syntax of the code, and more especially to the usage of the same variable names, it would perform badly if we give it in input a code with unknown tokens.

On the other hand, if the model performs well, even when the code is obfuscated, this could conduct to say that effectively the Seq2seq seems to **catch more the semantics** of the vulnerability than the syntax.

RQ3. Is Seq2seq sensitive to noise ? This question is about assessing if the model can perform well, even if noise (some useless lines) is inserted in the instances.

If the model is not disturbed by these lines, that are not related to the vulnerability itself, it shows that the Seq2seq model is able to **give attention to the correct part** of the code in order to inject vulnerabilities.

RQ4. How Seq2seq predictions differ depending on the variants of the dataset ? This question is about assessing a semantic difference between the predictions and the initial vulnerable methods for the different experiments (normal, obfuscated, augmented). This, in order to find if altering the initial dataset makes the model able to produce more "original" vulnerable methods.

RQ5. Are the predictions able to fool a static analyser ? This question is about finding if the Seq2seq would be able to create more subtle vulnerabilities that would fool Infer, i.e. making the tool consider a malicious code as benign.

In order to answer these questions, different experiment have been done with different alterations on the dataset. To evaluate these experiment, we performed 5 Fold cross validation on a subset of the Juliet dataset.

4.4 Seq2seq with Juliet dataset (5 CWE) Evaluation

With Infer able to detect some instances, a new dataset has been made with only the test cases for which Infer is able to dectect the vulnerability in the vulnerable method and detect nothing in the goodB2G methods (because we will work only on these ones).

Goal The goal of these experiments is to analyse and assess the Seq2seq model according to certain manipulations on the data.

Metrics For this purpose we want to compute different metrics at different iterations. As the model seems to reach its best accuracy after 10000 iterations, the following metrics will be computed at 2000, 6000 and 10000 iterations.

- The ratio of compilable predictions.
- The ratio of exact matching between the prediction and the initial vulnerable method of the test case.
- The ratio of vulnerability inserted.

.

Note: Sometimes the vulnerability is not detected as usually (Infer set another type of vulnerability for the same case). For these files, a manual analysis of the Infer result has been made to be sure of the classification (as explain during the Section 3.4.2 talking about the notebooks).

Note 2: For these calculation, only the first beam predicted will be selected. We saw during the previous experiments that it is most of the time the best one. But it can happen that the vulnerable version of the instances is another beam result.

Hyperparameters The selection of the hyperparameters is a mix between what we can see on the state of the art and the hyperparameter optimisation.

- RNN size: 256 (as in [10]).
- Batch size: 32 (as in [8]).
- Word vec size: 256 (as in [8]).
- Vocab size: 250 (1000 in [8], 129 in [35] and around 430 in the work of and Tufano et al. [24].
- Two layers for the encoder and the decoder.
- Bidirectional LSTM for the encoder and simple LSTM for the decoder.
- Attention and Copy mechanisms.

As we want to be more precise and reduce the possibility to have ideal instances in the test set, we used a 5 Fold Cross Validation to assess the performances of the model. Then, at the end of the 5 experiences, the average metrics performances will be computed.

Find the appropriate limit of tokens When we started this experience, we filtered the dataset only on the number of different lines set to a maximum of 10. Surprisingly, we got really poor results comparing to the experience with maximum of 6. This is at this moment that we decided to find if there exists a kind of maximum of tokens limit from which the model is not able anymore to produce compilable predictions.

To achieve that, we created a script and analysed the number of tokens of the initial vulnerable method for the predictions compilables vs not compilables.

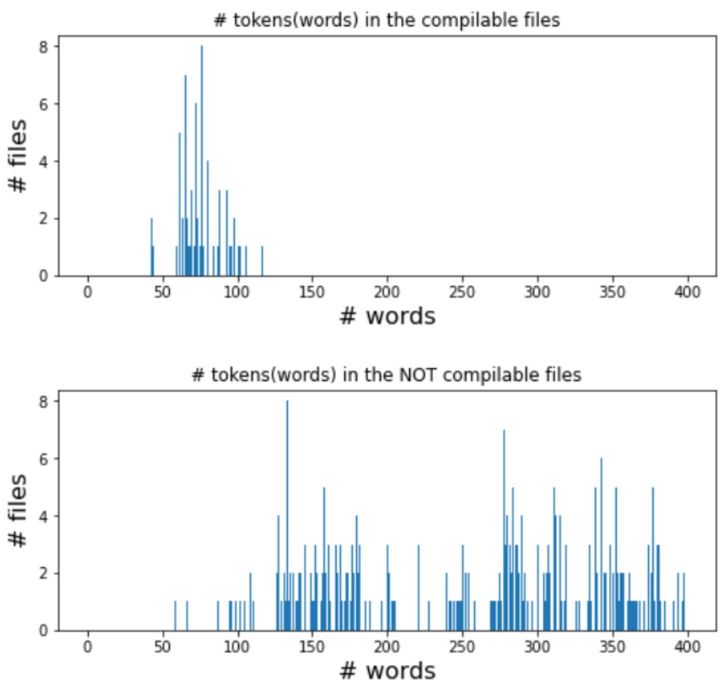


Figure 17: Number of tokens compiled vs not compiled files

As we can see on Figure 17, the model seems not able to compile if the vulnerable method is supposed to be a bit more than 100 tokens.

Then, we decided to filter the dataset only on the number of tokens fixed to 150. We choosed 150 instead of directly 100 in the case the dataset alteration would raise the performance of the model in terms of tokens it is able to correctly predict.

4.4.1 Normal dataset

The first experiment is with the normal dataset. That means we didn't alter the test cases.

Evolution of the data From the 4060 test cases results 6991 instances but after the token filtration, 1347 instances are remaining. And from these latest filtered on the identical benign methods, 1275 instances are remaining.

As the K Fold Cross validation is set on K = 5, the number of instances per fold is equal to 255.

Final results The results computed on the whole 5 Folds can be seen on Table 3.

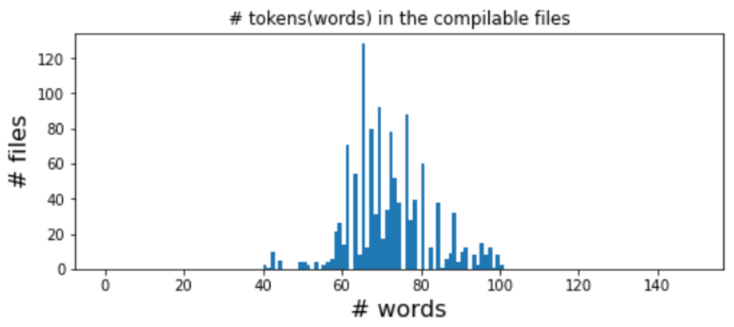
Metric \ Iteration	2000	6000	10000
% compiled prediction	93%	94%	94%
% exact matching	90%	92%	93%
% vulnerability inserted	90%	92%	93%

Table 3: Results of the normal dataset experience.

Analysis We can see that most of the files are compilable and that most of them are a flawed version of the method. We can see a direct link between the number of exact matching with the initial vulnerable method and the vulnerability inserted. That means that the model is able, most of the time, from an unseen benign method to predict the initial vulnerable method.

A really few number of predictions were not exactly matching the initial vulnerable method (so few that it does not even appear on the ratio). But for these ones, a vulnerability was still inserted. After a manual review it can happen that an instruction is modified. Like instead of fixing a value of data in a never explored else branch, it predicted a dynamic assignation. These predictions will be analysed thanks to SootDiff later in this document.

Is it still the 100+ tokens that don't compile? If we look at the graph on Figure 18 for the compilation according to the number of tokens for the iteration 10000, we can see again the separation between the methods at approximately 100 tokens. Hence our model is still blocked at this limit.



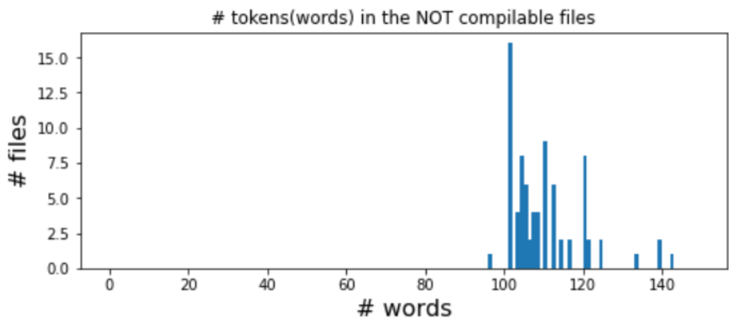


Figure 18: Normal dataset - # tokens in compiled vs not compiled files for the iteration 10000 over all the Folds

4.4.2 Obfuscated dataset

This experiment concerns the usage of the obfuscated dataset. Once the 4060 test cases have been obfuscated it was necessary to verify that Infer can still discover the vulnerabilities. This was effectively the case so it was correct to use this dataset.

The advantage of this one is that the variable names are different, then if the model is really sensitive to the syntax, it should be totally disturbed by this alteration and the metrics should not be as good as in the normal dataset. On the other hand, if the model is, at least a little, able to catch the semantics, it should not be disturbed as much. This is what we will try to discover with this experiment.

Evolution of the data From the 4060 test cases results 6991 instances but after the token filtration, 1347 instances are remaining. From these latest filtered on the identical benign methods, 1347 instances are remaining. This is normal due to the obfuscation, the previous same benign methods are not syntactically the same anymore, the name of the variables and of the Strings has changed.

As the K Fold Cross validation is set on K = 5, the number of instances per fold is equal to 269 or 270.

Final results The results computed on the whole 5 Folds can be seen on Table 3.

Metric \ Iteration	2000	6000	10000
% compiled prediction	89%	90%	90%
% exact matching	16%	14%	16%
% vulnerability inserted	89%	89%	90%

Table 4: Results of the obfuscated dataset experience.

Analysis As we can see, the results are a bit less good than the experiment with the normal dataset and the number of exact matching plummets compared to the results with the normal dataset. This already provides a first impression that, effectively, the model catches the semantics. It is then necessary to understand which files didn't compile anymore and why.

Is it still the 100+ tokens that don't compile? If we look at the graph on Figure 19 for the compilation according to the number of tokens for the iteration 10000, we can see now some changes. We still see the files are not compilable beyond 100 tokens. But some files under 100 tokens are now not compilable.

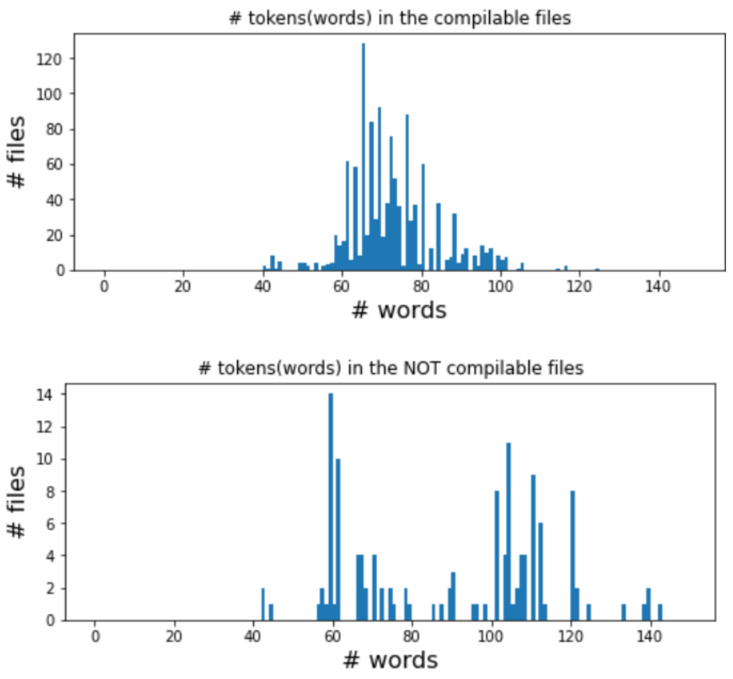


Figure 19: Obfuscated dataset - # tokens in compiled vs not compiled files for the iteration 10000 over all the Folds

After a manual analysis most of them are the xxxx17.java test cases. These ones contain a for loop and the copy mechanism seems not to work for these. There is an example of a loop predicted by the model.

```
for (int <unk >= 0; <unk > <1; <unk > ++){
    ...
}
```

The code above can not compile because it is not syntactically correct. We tried to force the model to replace the unk token with the argument *-replace_unk*, but the results on one experiment were bad too. Then we assumed that done this experiment with this argument again wouldn't raise the performance of the model using this dataset.

Without this small problem, the performance of the model with these data would be really close to the performance with the normal dataset.

Why so few exact matching ? The reason of this is due to a syntactic difference between the predicted vulnerable version of the method and the initial one. But this difference can be semantic too (e.g. a modification in a if condition).

If this is due to a semantic difference, this will be catched by SootDiff at the moment of the SootDiff analysis Section. Concerning the syntactic difference, we can already know that this is due to the fact that Strings contain ASCII codes in the new dataset. Hence it often happens that there are differences in the Strings between the prediction and the initial vulnerable method resulting in a “not exact matching”.

Does this model needs more training to compile more? We could maybe think that this model need more train iteration due to an increased complexity of the dataset. But it doesn’t seem to bring much improvement. The experiment done on the fold0 results in one additional compilation when going from 10000 to 20000 iterations.

Bias ? This dataset could be biased by the fact that some semantically similar instances of the benign methods are in the dataset. Indeed, no equivalent benign version have been removed. But we know that if the dataset had not been obfuscated we should have removed some of them.

Then, if the model is able to catch more or less the semantics (the behaviour of the vulnerability), that would mean that this experiment trains on some same semantic instances. And this can introduce a bias in the test set result. Indeed, the model would be able to predict the test set instances more easily because it will have seen semantically similar instances during training.

Then another experiment has been made, removing these potentially similar instances in order to see if this current experiment was helped to insert vulnerabilities.

4.4.3 Obfuscated v2 dataset

In order to remove a potential bias, this experiment is about using the same test cases as the ones used in the Normal dataset. This will ensure that there are no initially identical benign methods in the dataset.

Evolution of the data From the 4060 test cases results 6991 instances but after the token filtration, 1347 instances are remaining. And from these latest filtered on the identical benign methods, 1275 instances are remaining.

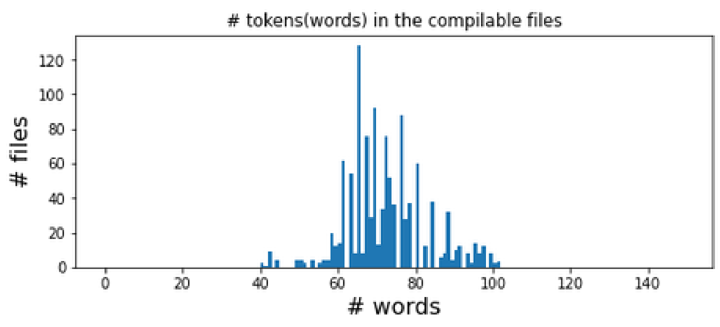
As the K Fold Cross validation is set on K, the number of instances per fold is equal to 255.

Final results The results computed on the whole 5 Folds can be seen on Table 5.

Metric \ Iteration	2000	6000	10000
% compiled prediction	89%	89%	90%
% exact matching	12%	16%	16%
% vulnerability inserted	87%	87%	88%

Table 5: Results of the obfuscated v2 dataset experience.

Analysis As we can see, the results are slightly worse than the first experiment with the obfuscated dataset. But we can have the same observation about the tokens on the Figure 20. We can think this is because this experiment is not biased compared to the previous experiment. This will be discussed further in the Research Questions Discussion.



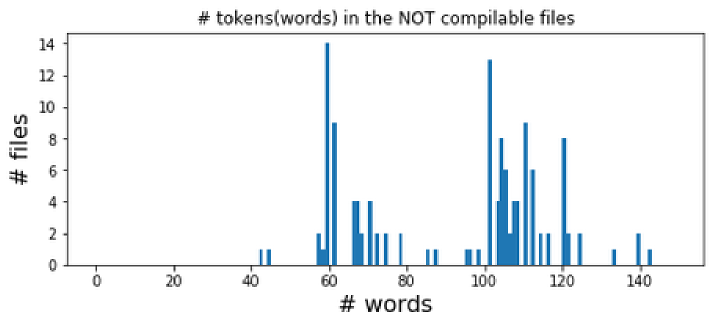


Figure 20: Obfuscated v2 dataset - # tokens in compiled vs not compiled files for the iteration 10000 over all the Folds

4.4.4 Mutants dataset

This experiment concerns an augmented dataset with some noise inserted inside the instances. It will show us if the model is sensitive to some useless lines of code inserted. Moreover, the code is partially abstracted and modified due to the passage into the compile - decompile process in the process of Confuzzion.

Evolution of the data From the 4060 test cases results 2227 instances after the data augmentation (so a raise of + 75% compared to the normal dataset). As the K Fold Cross validation is set on K = 5, the number of instances per fold is equal to 445 or 446.

Let's describe the different results in term of test cases during the augmentation process.

Augmentation with Confuzzion As we can see on Table 6, asking Confuzzion to give us 50 mutants per seed file gives us 28426 compilable mutants (including the seed transformed through the compile-decompile process). But finally only 21487 test cases will be kept.

Test cases \ CWE	129	134	190	191	369	Total	↑
# Seed files	1116	162	1374	1372	36	4060	100%
# Compilable mutants	3660	1714	15955	7061	36	28426	700%
# Accepted by Infer mutants	2348	1424	13407	5009	36	22224	547%
# Final mutants	2230	1154	13169	4898	36	21487	529%

Table 6: Augmentation with Confuzzion.

During the process of filtering, we use Infer to filter the test cases. But it appeared that some seeds where unaccepted by ant (that is the method used with Infer to compile). Hence we had to remove them. The result of this operation corresponds to the "Accepted by Infer Mutants".

Finally, the "Final mutants" corresponds to the mutants where Infer detects the vulnerability in the vulnerable method and nothing in the goodB2G methods.

Then, it is necessary to go through the Data Preparation process (that is summarised in the next subparagraph).

Filtration of the data Now that we have our final test cases, it is necessary to prepare and filter them. In order to have an idea of the augmentation, we will compare the number of instances, at each step, for this dataset versus the normal dataset.

Filter \ dataset	Normal	Mutants
# Test cases	4060	21487
# Instances	6991	39464
# Instances with # tokens <= 150	1347	15007
# Instances unique	1275	2227

Table 7: Data Preparation of the mutants.

From the results of the Table 7, we can see that starting from an augmentation of +429%, we only keep an augmentation of +75% after the preparation of the data. This is normal and directly due to the filtering of maximum number of tokens. Indeed, the process of creating mutants raised the size of the methods, but we are filtering on this size so this is an expected behaviour .

Problem encountered During the augmentation process, the seed is going through the compile-decompile process. Thus, the compiler sometimes modifies and compacts the code. The result is the same semantic code but the syntax is a bit different. At this moment, we discovered some problem with the untokenisation process. The "&&" tokens where separated by a whitespace, so the compiler didn't compile these files and the chars (e.g. ' 0 ') were not accepted if there is was a whitespace between the character and the quotes. Hence we had to apply some updates to this process.

Moreover, this makes us able to discover that going through this process is a good idea to obtain new instances semantically identical and syntactically different. There is an example of what we could see:

```
// Conditions in the initial file
if (privateFive==5)
{
    if (data != null)
    {...

// Conditions in the new file
if (this.privateFive == 5 && var1 != null
    ...
```

From this, we can discover a new operator that our model never faced before, the "&&". We can see the variable "data" becoming "var1", that's the phenomenon we speak about by partially abstraction of the code. Because the variable names are modified but are often the same (where in the obfuscation, the data variable was, almost every time, different).

Now the process of data augmentation is well defined, let's talk about the results.

Final results The results computed on the whole 5 Folds can be seen on Table 8.

Metric \ Iteration	2000	6000	10000
% compiled prediction	68%	69%	69%
% exact matching	65%	68%	68%
% vulnerability inserted	67%	69%	69%

Table 8: Results of the mutants dataset experience.

Analysis The results seem lower than the normal and obfuscated experiments, but this is probably due to the different distribution of the files in terms of number of tokens.

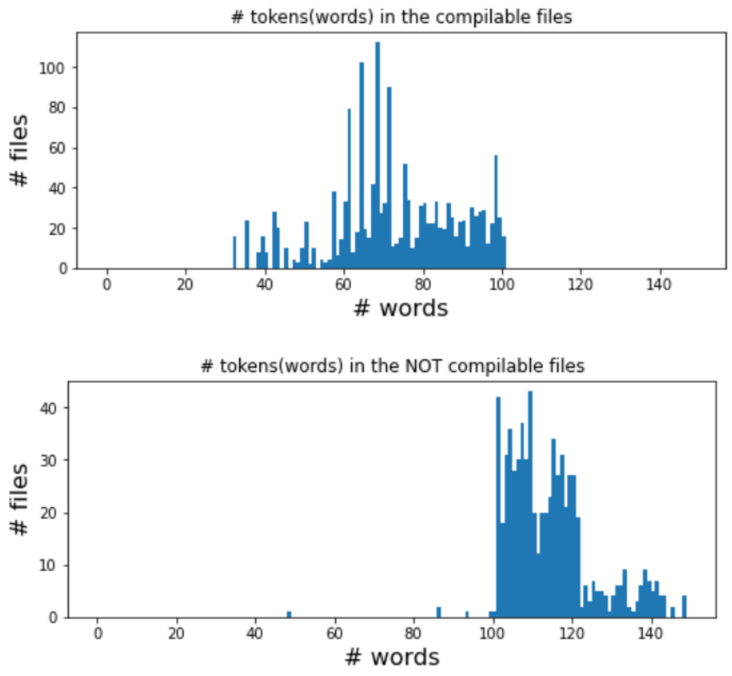


Figure 21: Mutants dataset - # tokens in compiled vs not compiled files for the iteration 10000 over all the Folds

This is what we can see on Figure 21. Almost every prediction less than 100 tokens is compilable. Then, the model seems not the suffer from noise in the dataset.

Up to now we found that altering the dataset seems not to enhance the abilities of the model to predict sentences longer than 100 tokens. Then we need to assess the performances of the model on equal terms. This is summarised in the following subsection.

4.5 Results and Discussion

RQ1. How effective is Seq2seq to inject vulnerabilities ? Table 9 shows the results of the different experiments. From what we can see, most of the compiled predictions contain a vulnerability. At a first look we could say that training the model on the initial dataset is the best way to inject vulnerabilities.

	Normal		Obfus.		Obfus. v2		Mutants	
test cases	1275	100%	1347	100%	1275	100%	2227	100%
compiled predictions	1195	93.73%	1212	89.98%	1146	89.88%	1531	68.75%
vulnerabilities inserted	1184	92.86%	1206	89.53%	1121	87.92%	1526	68.62%

Table 9: General experiments comparisons at iteration 10000.

However, the different experiments showed that, even when we modify the dataset, the model is not able to easily predict compilable methods upon 100 tokens (see Figures 18, 19, 20, 21).

Then, to be able to make better conclusion, it is necessary to filter on the instances where the method supposed to be predicted is < 100 tokens. This is what we can see on the Table 10.

	Normal		Obfuscated		Obfuscated v2		Mutants	
test cases	1194	100%	1256	100%	1194	100%	1520	100%
compiled prediction	1193	99.92%	1190	94.75%	1141	95.56%	1512	99.67%
vulnerability inserted	1182	98.99%	1184	94.27%	1116	93.47%	1510	99.34%

Table 10: Experiments comparisons at iteration 10000 where the initial vulnerable method <100 tokens.

The interpretation of the result is then different. The experiments with the normal dataset and the mutants get really good results but we can see a lower result when the dataset is obfuscated.
This can be explained by difficulties of the copy mechanism to find the value in the for loops (test case n° 17). This represents 51/66 of the non-compilable predictions for the obfuscated v1 experiment and 49/53 for the v2 experiment.

Performances on another dataset In order to assess the general performances of this model, it would be great to assess its capability on another dataset. That's why we decided, in order to get a first idea, to choose the methods from the ArrayUtils library.

Unfortunately the results were really bad. That means the model is limited to this dataset, or at least methods really similar to what we can see on this dataset.

This is a pity because if the model was able to predict, even rarely, something exploitable, it would have been possible to compare its performances according to the alterations that the training dataset would have received. This could have shown that training the Seq2seq on obfuscated data makes it more likely to introduce vulnerabilities in code that is outside of what it has seen during training for example.

RQ1. The Seq2seq gets good results to inject vulnerabilities. Most of the predictions are compilable and contain a vulnerability, especially when the prediction is less than 100 tokens. Unfortunately, the capability of vulnerability injection seems to be limited to this dataset and then the difference in performance depending on the alterations made to the dataset cannot be evaluated.

RQ2. Is Seq2seq sensitive to the usage of the same variable names during the training ? If the model was really too sensitive to the syntax, it would have been unable to provide correct predictions for the experiments with the obfuscated datasets. Indeed, the model would be lost when facing a new variable name and then wouldn't be able to predict accurately the complete sentence.

Finally, the results obtained may allow us to say that the model captures, in some way, the important elements of the method in order to be able to insert a vulnerability. This is certainly due to the attention mechanism, which when helped by the copy mechanism, can, most of the time, succeed in inserting the vulnerability correctly.

RQ2. No, the Seq2seq model with the Attention and Copy Mechanisms is insensitive to the usage of the same variable names during the training. It manages to capture the semantics of the vulnerability to inject it in a benign code.

RQ3. Is Seq2seq sensitive to the noise ? The results obtained on the experiment with the augmented dataset are really high and close to those with the normal dataset. This may allow us to say that effectively the model is able to pay attention to the right parts of the code in order to inject vulnerabilities.

RQ3. No, the Seq2seq model with the Attention and Copy Mechanisms is insensitive to the instructions not relative to the vulnerability. It manages to pay attention at the right part of the code to inject a vulnerability.

RQ4. How Seq2seq predictions differ depending on the variants of the dataset ? In order to find if the Seq2seq is able to predict more creative vulnerable methods, we assessed the distance between the initial vulnerable method and the predicted one at the bytecode level. We want to find the real semantic differences, like the modification of an "if" condition, but we faced an issue with the Strings.

Problem: Untokenisation process for Strings In order to compute the semantic distance in the predictions, we had to make a choice due to the untokenisation process and the dataset.

Indeed, all the test cases in the dataset do not have the same whitespaces at the same place. for example we can sometimes find “result :” and sometimes “result:”. This is not a semantic difference from our point of view, but this is seen as a difference by SootDiff. Moreover, the “<unk>” tokens that appears in the String will lead to an augmentation of the metric distance, but this is not a significant semantic difference.

Hence, multiple choices came to us:

- Do not take care to this and still compute the metric. This was the easiest solution but then we do not know from which Levenshtein distance starting to take into account that this is a real semantic difference between the two files.

- Remove every whitespaces in the Strings in both the initial dataset and the test cases with the prediction. This was a good idea but then the <unk> tokens would still be taken into account.

- Do not compute the metric on the instructions with certain keywords. This was not easy to find keywords appearing when two strings are detected as different without side effects on the other instructions.

- Do not touch at the files, run SootDiff and create a regex removing the content of every Strings from the SootDiff reports. This is an inconvenient for the CWE134, because these ones insert vulnerabilities by modifying Strings but the predictions always have vulnerability inserted (except for 3 of them for which no vulnerability have been inserted) and as the only way to introduce vulnerability on the CWE is by setting the String to the correct value, but we are not especially looking for creativity inside this String. Hence this solution was selected.

Results As usually, a Notebook was made for this purpose. It takes the idea of the Notebook analysing the Infer results, for giving it the information about which prediction is detected as vulnerable. Then, gathering the information from Infer and SootDiff makes us able to find the predictions semantically different from the original and if they are, or not, vulnerable.

The following results represent the percentage of predictions with a Levenshtein distance $>$ than 0. The percentage is then related to the percentage of non-exact matching without, normally, taking into account the non-exact matching concerning the Strings.

Compilable prediction These results concern the percentage of prediction with a Levenshtein distance > 0 without taking care of if there is a vulnerability inserted or not.

Dataset \ Iteration	2000	6000	10000
Normal	3.21%	1.67%	1.09%
Obfuscated	4.69%	5.34%	5.20%
Obfuscated v2	2.63%	2.91%	2.36%
Augmented	4.02%	0.97%	0.33%

Table 11: Percentage of prediction with a semantic difference compared to the real vulnerable method.

We can see on Table 11 that the experiments with the obfuscated dataset are the most different. This can be seen on Figure 22 where the experiment with this dataset gets the most instances with a difference and they are, most of them, around 200 edit distances.

But this may be because they are not able to insert a vulnerability, so we can filter these results for the test cases with a vulnerability inserted only.

Compilable prediction with vulnerability These results concern the percentage of prediction with a Levenshtein distance > 0 when a vulnerability is really inserted.

Dataset \ Iteration	2000	6000	10000
Normal	0.51%	0.00%	0.25%
Obfuscated	4.69%	3.94%	4.46%
Obfuscated v2	0.44%	0.44%	0.35%
Augmented	2.83%	0.26%	0.00%

Table 12: Percentage of vulnerable prediction with a semantic difference compared to the real vulnerable method.

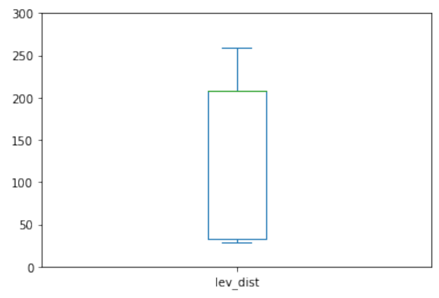
On Table 12 we can see that the percentage of test cases is lower for the obfuscated v2 but not the first one. That means, and confirms what we said before, that the experiment with the obfuscated dataset is more able to insert different vulnerability because it trains on "semantically identical" benign methods for different vulnerable methods. Then, it is able to predict another vulnerable method than the expected one when it faces one of these "duplicated" benign method.

This analysis confirms our hypothesis that the model is not especially sensitive to the syntax and is able to catch the semantics of the vulnerabilities to inject them in benign codes.

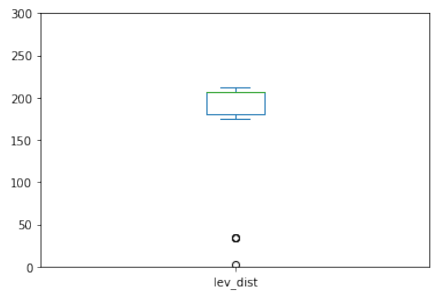
Beside, we can see that the Seq2seq seems not really able to predict creative vulnerable methods. This was already our intuition when we saw the percentage of exact matching for the experiments with the normal and augmented dataset. But this experiment was great to confirm that the model is not sensitive to the syntax of the code (in terms of name of the variables and Strings).

RQ4. The predictions of the Seq2seq are rarely different from the initial vulnerable method except when the model trains on instances where the benign methods are semantically identical. This reinforces the idea that the model captures the semantics of the vulnerability and is not sensitive to syntax.

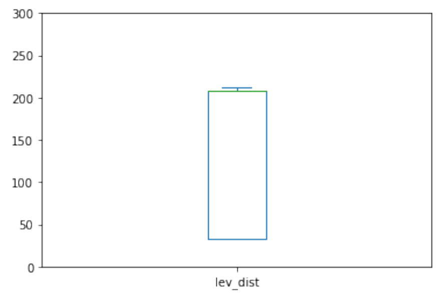
RQ5. Are the predictions able to fool a static analyser ? To answer this question, we manually looked at every compiled prediction where Infer didn't find the vulnerability. Most of the time, the test case corresponds to the CWE129 for which the Seq2seq didn't modified correctly the if condition. Indeed, the modification of this condition could lead to a vulnerability if the model, in addition to that, modified the value of the variable corresponding to the index.



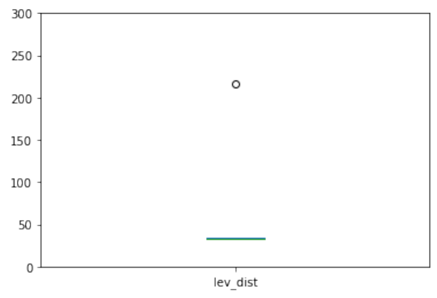
(a) Normal dataset: 13 instances



(b) Obfuscated dataset: 63 instances



(c) Obfuscated v2 dataset: 27 instances



(d) Mutants dataset: 5 instances

Figure 22: The comparison of the Levenshtein values for the iteration 10000 of the experiments

Hence, finally, it seems that every predicted method where Infer did not detect a vulnerability does really not contain a vulnerability.

RQ5. The Seq2seq seems not able to create more subtle vulnerabilities that could bypass Infer.

4.6 Threats to validity

During the progress of the experiments, we have noted different threats to the validity of the experiments and analyses.

4.6.1 Internal validity

The possibility to replace the unk tokens During these experiments, we saw some <unk> tokens resolving in the impossibility to compile the predictions. This problem comes from the fact that the Copy Mechanism can not find the token to copy from the source. Then, in order to resolve this problem, we tried to use a specific parameter ("-replace_unk") during the translation that aims to force the model to replace this <unk> token by another one. This has been done for an experiment and the result was that the <unk> in the "for loop" was replaced by either "int" or "}". This wouldn't change any of the results. Moreover, all the other <unk> token seem to be inside String contents. These ones have no impact on the prediction. Hence we are confident to say that the usage of this argument wouldn't have changed the results and the analysis.

Tokenisation and untokenisation processes The tokenisation and untokenisation processes were done by hand with regex and Python. They are therefore potentially incomplete and some decisions had to be made.

It is not unlikely that splitting the tokens in another way could slightly modify the results. Indeed, we had to choose how and where to split the tokens.

Concerning the untokenisation process, we found that it was not always complete. The results were manually inspected during the experiments, so this process evolved step by step and finally was able to reassemble the tokens correctly. However it is not impossible that errors are still present.

This problem can be seen as a problem of the Seq2seq. This model does not explicitly know the rules for creating compilable code. Then we must take care of how we give it the tokens and how to transform predictions into compilable methods.

Hyperparameters settings Optimisation has been done but we saw that only look at the metrics is not enough to know if the model will predict correct predictions (in term of compilability and vulnerability inserted).

Some parameters may have been forgotten (e.g. the -replace_unk may have helped the model to predict more compilable solutions. But in this precise case we discovered that it doesn't).

Then add or modify some hyperparameters could maybe improve the results.

Other beams During the experiments, only the prediction with the best score was taken into account. This is because the best answer should be the first result, and during our first experiments we found that this was mainly the case. However, it is possible that sometimes, in the few predictions where no vulnerability was detected, that the correct answer is one of the other predictions with a worse score.

4.6.2 External validity

Model limited to specific dataset After a quick experiment on other methods coming from another dataset (ArrayUtils) we show the non-capability of the model to introduce the CWE vulnerabilities in another dataset.

To solve this point, it should be necessary to replace the RNN by a pre-trained model like CodeBERT that would be more able to deal with more "heteroclites" instances. The state of the art showed that the usage of this kind of model are more powerfull than simple RNN-based models [26].

5 Conclusion

To conclude, we propose different perspectives to improve the work done in this thesis.

First, it could be great to use more powerful models like Codebert. Indeed, these kind of models are really cost-effective to train, but it is possible to apply a Transfer learning technique on them in order to specialise them in vulnerability injection. Second, select other datasets, like the one used by LineVul and discovered at the end of this thesis [26].

During the different experiments and through the different Research Questions, this thesis makes several contributions.

Data Augmentation During the experiments, we showed different ways to augment a dataset. Whether through obfuscation, adding “noise” or through the compilation-decompilation process, it is possible to obtain a great number of additional instances with different syntax but identical semantics. These instances each have different characteristics and we hope that, in the future, the use of these techniques could improve the performance of a model.

NLP in Vulnerabilities Injection As explained in the Section 2 about the Background of this thesis, the NLP techniques have several advantages in Fault injection. The model automatically learns where and how mutate the code to inject a fault. This paper has shown that when focusing more on vulnerabilities, these models can also have their place.

However, it is necessary to have a large amount of data available, or at least to use more powerful sequence-to-sequence models to inject vulnerabilities into any code.

As explained by von der Mosel et al., for NLP tasks that require SE context understanding, it is necessary to pre-train the model [36]. Therefore, we believe that a great potential future work would be to fine tune a pre-trained model like CodeBERT to make it capable of injecting vulnerabilities.

NLP and semantics of the code The main purpose of this thesis is to inject vulnerabilities into code in a semantic-way (we recall our definition in Section 1). To do so, we have chosen a model which, after having gone through the state of the art, is able, in a certain way, to understand the semantics.

During the experiments, we tried to evaluate whether or not this model seems to be able to capture the semantics. From what we have seen, whether it is thanks to the obfuscation in general or by passing from the experiment with the obfuscated dataset to the second version of this dataset, it seems to us that the Seq2seq is indeed able to capture important elements of the code in order to know where and how to modify the code to make it vulnerable. It is important to note that this one incorporates the copy and attention mechanisms.

Token limitation of the Seq2seq We demonstrated that, at least on this dataset, the Seq2seq from OpenNMT is not able to predict sentences of more than 100 tokens. However, this is probably due, at least in part, to the selection of the metaparameters. Unfortunately we did not find the reason of this limit.

To conclude, this thesis introduces the injection of security vulnerabilities in Java code thanks to NLP techniques. Starting from a corpora of well-known (CWE) vulnerabilities, we demonstrate that a quite simple model as the Seq2seq is able to learn, in a semantic-way, how to insert the vulnerability. Moreover we demonstrated that it is possible to modify the instances of the dataset to increase the training corpora and that in spite of this the model is able to focus on the main elements to insert a vulnerability. We consider these observations as a new proof to consider these techniques able to catch, in a certain way, the semantics of the code.

Unfortunately, the observations made during the quick experiments on new test sets tend to show that the model quickly loses performance when it has to insert vulnerabilities in code that is very different from what it encountered during the training. This is why we think that to create a much more powerful tool, it would be wise to use much larger models such as CodeBERT. The usage of a model such as this one should allow the tool to insert vulnerabilities in much more heterogeneous methods.

References

[1] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, “Towards security-aware mutation testing,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 97–102, 2017.

[2] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “Lava: Large-scale automated vulnerability addition,” in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 110–121, 2016.

[3] V. Kashyap, J. Ruchti, L. Kot, E. Turetsky, R. Swords, S. A. Pan, J. Henry, D. Melski, and E. Schulte, “Automated customized bug-benchmark generation,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 103–114, 2019.

[4] P. Black, “Juliet 1.3 test suite: Changes from 1.2,” 06 2018.

[5] Semantic Designs, “Java source code obfuscator.” `http://www.semdesigns.com/products/obfuscators/JavaObfuscator.html`.

[6] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, “Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities,” pp. 586–597, 12 2021.

[7] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving fast with software verification,” in *NASA Formal Methods* (K. Havelund, G. Holzmann, and R. Joshi, eds.), (Cham), pp. 3–11, Springer International Publishing, 2015.

[8] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[9] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, and Y. L. Traon, “Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach,” *arXiv preprint arXiv:2012.11701*, 2020.

[10] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Learning how to mutate source code from bug-fixes,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–312, 2019.

[11] J. A. Clark, H. Dan, and R. M. Hierons, “Semantic mutation testing,” *Science of Computer Programming*, vol. 78, no. 4, pp. 345–363, 2013. Special section on Mutation Testing and Analysis (Mutation 2010) & Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing.

[12] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–10, 2015.

[13] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyand'e, J. Klein, and Y. L. Traon, "Ibir: Bug report driven fault injection," *ArXiv*, vol. abs/2012.06506, 2020.

[14] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 224–234, Association for Computing Machinery, 2018.

[15] J. Pewny and T. Holz, "Evilcoder: Automated bug insertion," *CoRR*, vol. abs/2007.02326, 2020.

[16] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies," *arXiv preprint arXiv:2112.14508*, 2021.

[17] M. Jimenez, T. Checkam, M. Cordy, M. Papadakis, M. Kintis, Y. Le Traon, and M. Harman, "Are mutants really natural? a study on how "naturalness" helps mutant selection," pp. 1–10, 10 2018.

[18] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," 2022.

[19] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, (New York, NY, USA), p. 906–918, Association for Computing Machinery, 2021.

[20] A. Amidi and S. Amidi, "Recurrent neural networks cheatsheet." `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks`.

[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[22] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[23] T. Keldenich, "Encoder decoder what and why ? – simple explanation." `https://inside-machinelearning.com/en/encoder-decoder-what-and-why-simple-explanation/`, 10 2021.

[24] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, sep 2019.

[25] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 505–509, 2021.

[26] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," 03 2022.

[27] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 336–347, 2021.

[28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[29] MITRE Corporation, "The Common Weakness Enumeration (CWE) Initiative." `http://cwe.mitre.org/`, 5 2022.

[30] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61, 2010.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[32] M. Alqaradaghi, G. Morse, and T. Kozsik, "Detecting security vulnerabilities with static analysis – a case study," *Pollack Periodica*, 2021.

[33] J. Brownlee, "A gentle introduction to k-fold cross-validation." `https://machinelearningmastery.com/k-fold-cross-validation/`, 8 2018.

[34] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[35] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, p. 1345–1351, AAAI Press, 2017.

[36] J. von der Mosel, A. Trautsch, and S. Herbold, "On the validity of pre-trained transformers for natural language processing in the software engineering domain," *arXiv preprint arXiv:2109.04738*, 2021.