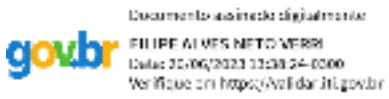
Dissertation presented to the Instituto Tecnológico de Aeronáutica, in partial fulfillment of the requirements for the degree of Master of Science in the Graduate Program of Electronics and Computer Engineering, Field of Informatics.

Leonardo Silveira

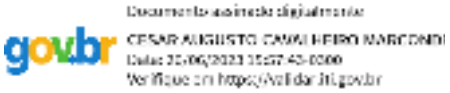
SOURCE CODE VULNERABILITY DETECTION AND INTERPRETABILITY WITH LANGUAGE MODELS

Dissertation approved in its final version by signatories below:



Prof. Dr. Filipe Alves Neto Verri

Advisor



Prof. Dr. Cesar Augusto Cavalheiro Marcondes

Co-advisor

Prof[a]. Dr[a]. Emília Villani

Pro-Rector of Graduate Courses

Campo Montenegro
São José dos Campos, SP - Brazil
2023

Cataloging-in Publication Data
Documentation and Information Division

Silveira, Leonardo
Source Code Vulnerability Detection and Interpretability with Language Models / Leonardo Silveira.
São José dos Campos, 2023.
60f.

Dissertation of Master of Science – Course of Electronics and Computer Engineering. Area of Informatics – Instituto Tecnológico de Aeronáutica, 2023. Advisor: Prof. Dr. Filipe Alves Neto Verri. Co-advisor: Prof. Dr. Cesar Augusto Cavalheiro Marcondes.

1. Deep Learning. 2. Transformers. 3. Vulnerability Detection. 4. Interpretability. I. Instituto Tecnológico de Aeronáutica. II. Title.

BIBLIOGRAPHIC REFERENCE

SILVEIRA, Leonardo. **Source Code Vulnerability Detection and Interpretability with Language Models**. 2023. 60f. Dissertation of Master of Science – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

Leonardo Silveira
Rua Jacob Vontobel 350
90690-080 – Porto Alegre–RS

SOURCE CODE VULNERABILITY DETECTION AND INTERPRETABILITY WITH LANGUAGE MODELS

Leonardo Silveira

Thesis Committee Composition:

Prof. Dr.	Lourenço Alves Pereira Júnior	President	-	ITA
Prof. Dr.	Filipe Alves Neto Verri	Advisor	-	ITA
Prof. Dr.	Cesar Augusto Cavalheiro Marcondes	Co-advisor	-	ITA
Prof. Dr.	Carlos Henrique Quartucci Forster	Member	-	ITA
Prof. Dr.	Zhao Liang	Member	-	USP

ITA

Acknowledgments

This journey had the participation of many smart, capable, and kind people, to whom I send my most sincere gratitude and appreciation.

In particular, I dedicate this accomplishment

To my advisor Filipe, who guided me to this point, always showing the highest degree of professionalism and care.

To my co-advisor Cesar, who accompanied and mentored me at every step of this work.

To my parents and sister, for being such great role models and providing me with all the support and love someone needs.

To my girlfriend Tamires, who was always by my side, even when distant.

And to Enola, who was my joyful partner in the last six months of this work.

"Never trust anything that can think for itself
if you can't see where it keeps its brain."
— Arthur Weasley

Resumo

A execução de testes de segurança em software para detecção de vulnerabilidades é fundamental de modo a evitar a ocorrência de ataques mal intencionados, que podem, entre outras coisas, comprometer o funcionamento da aplicação ou expor dados sensíveis de seus usuários. Tradicionalmente, esses testes são realizados utilizando ferramentas de análise estática, dinâmica ou simbólica. Essas ferramentas apresentam diversas limitações, como a detecção de muitos falsos positivos e custo computacional muito elevado. Uma alternativa aos métodos tradicionais é o uso de aprendizado de máquina, particularmente modelos de aprendizado profundo. Diversas abordagens explorando esses modelos foram propostas na literatura recente, sendo sua maior parte inspirada nos avanços alcançados em aprendizado de máquina profundo na área de Processamento de Linguagem Natural. Apesar do uso de modelos cada vez mais sofisticados, pouco foi feito no campo de interpretabilidade desses métodos, com o objetivo de alcançar não somente a classificação do código fonte como vulnerável ou não, mas também apontar no código os trechos em que o modelo de aprendizado acredita que a vulnerabilidade está presente. Nesse trabalho, realizamos o treinamento (*fine-tuning*) de dois modelos de linguagem, CodeBERT e CoTexT, para a tarefa de detecção de vulnerabilidades em códigos fontes, e avaliamos a capacidade desses modelos de gerarem interpretação de suas predições. Para isso, realizamos a curadoria de um banco de dados composto por uma coleção de códigos vulneráveis e suas respectivas máscaras de rotulagem, localizando a posição exata das vulnerabilidades no código. Utilizando os dois métodos de interpretabilidade mais indicados para tarefas de classificação de texto em Processamento de Linguagem Natural, Saliência e InputXGradient, geramos mapas de calor representando a importância de cada token do código para predição do modelo. Com isso, encontramos que ambas as técnicas apresentam resultados de precisão semelhantes, porém os mapas de calor gerados pelo método InputXGradient são consideravelmente mais fáceis de serem interpretados. Comparando os modelos de linguagem, o CodeBERT apresenta acurácia levemente inferior quando comparado ao CoTexT na tarefa de classificação de códigos fontes, porém em contraste, gera resultados consideravelmente melhores de interpretabilidade de sua predição. Adicionalmente, destacamos o efeito dos tokenizadores utilizados pelos modelos de linguagem em sua capacidade de gerar interpretação de suas predições, mostrando que suas características podem influenciar de

maneira importante a habilidade do modelo de aprender a estrutura sintática e semântica da linguagem.

Abstract

The probing of software by security testers to detect possible vulnerabilities is of primary importance to prevent attackers from exploiting these flaws, which can, besides other things, compromise the application and permit access to sensitive data. Traditionally, these security testers are static, dynamic and symbolic analyzers. These tools, in general, present a compromise between precision and complexity. A recent alternative to these methods is the use of machine learning models, which can avoid this tradeoff altogether, rendering precise predictions with acceptable complexity. Recently, many models have been proposed in the literature, drawing much inspiration by the recent advances in the field of Natural Language Processing. Despite this recent exploration of machine learning methods, there is little study in the realm of interpreting these models' output, with the goal of achieving not only the classification of whether a snippet of code is vulnerable or not, but to be capable of pointing in the code where the model believes the vulnerability is located. In this work, we fine-tuned two state-of-the-art large pre-trained language models, CodeBERT and CoTexT, for the task of vulnerability detection in programming language code, and evaluated the ability of these models to render interpretation of their output. For this, we curated a benchmark dataset composed of a collection of vulnerable code and their respective ground-truth masks, locating exactly where the vulnerability is present in the code. Using the two best performing interpretability methods for text classification in the Natural Language Processing literature, Saliency and InputXGradient, we generated heatmaps with the importance of each input token to the final prediction. We found that both methods give similar precision results, but InputXGradient heatmaps are considerably easier to interpret. Comparing the language models, CodeBERT presents a slightly inferior classification accuracy compared with CoTexT, but contrastively, the former can give considerably better interpretation results for its decision. Additionally, we highlight the effect the tokenizers have on the models' ability to generate interpretation of their predictions, showing that the tokenizer characteristics can strongly affect the model's abilities to learn the syntactic and semantic structure of the language.

List of Figures

List of Tables

List of Abbreviations and Acronyms

MLP	multi-layer perceptron
NLP	natural language processing
DL	deep learning
NL	natural language
PL	programming language
CNN	convolutional neural network
RNN	recurrent neural network
GNN	graph neural network
AI	artificial intelligence

List of Symbols

w	Weight
b	Bias
V	Value Matrix
Q	Query Matrix
K	Key Matrix
X	Matrix of Input Vectors
d	Dimensionality
$Softmax$	Softmax non-linearity

Contents

1 Introduction

Probing software for security flaws is paramount to prevent attackers from exploiting possible vulnerabilities, which could lead to compromising software applications, accessing sensible data, and jeopardizing its users. Testing for security vulnerabilities becomes even more critical due to the widespread use of open-source libraries, which can propagate the impact of hidden vulnerabilities to seemingly secure software.

Traditionally, one way to discover code vulnerabilities is through the use of software static testing, which executes a set of pre-defined rules against the source code without actually needing to execute the program (RUSSELL *et al.*, 2018). More comprehensive approaches to software probing are dynamic and symbolic testing, where the program is executed repeatedly and all the feasible execution paths are evaluated. The major drawback of this latter class of methods is that the run-time and complexity of execution are often prohibitive (YAMAGUCHI *et al.*, 2014). Additionally, graph-based approaches leveraging the code structure, such as Abstract Syntax Tree (AST), were proposed (YAMAGUCHI *et al.*, 2014). However, the downside of graph-based methods is that they require the complete project to be compiled, which may be expensive or even not possible when only part of the code is available, as in a pull request, for instance (BURATTI *et al.*, 2020).

Recently, effort has been made to apply data-driven techniques, especially deep learning methods, to programming language understanding tasks, such as vulnerability detection. The main advantage of these methods is that they alleviate the need of extracting hand-crafted features from code or applying a set of rules designed by experts. In contrast, these methods learn code features that enable them to detect vulnerabilities directly from a labeled dataset of source code samples, in an end-to-end fashion (RUSSELL *et al.*, 2018).

Work applying deep learning to vulnerability detection has primarily employed techniques commonly used in Natural Language Processing (NLP) tasks, such as 1-dimensional Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) (RUSSELL *et al.*, 2018), and more recently large pre-trained Transformer language models (BURATTI *et al.*, 2020; FENG *et al.*, 2020). Additionally, some authors have leveraged the code topology and structure, such as its AST, data and control flows graphs, to apply Graph Neural Networks (GNN) for code classification (ZHOU *et al.*, 2019; NGUYEN *et al.*, 2021).

The Devign dataset (ZHOU *et al.*, 2019) is the main benchmark for the task of vulnerability detection. It consists of roughly 20,000 C/C++ functions hand-labeled as vulnerable or not vulnerable, with a balanced distribution between both classes. Models trained on this dataset are able to predict the function classification, but they are not able to give further information on the nature of the vulnerability or where in the code it is located. Additionally, the state-of-the-art model in this benchmark presents an accuracy of 66.7% (LU *et al.*, 2021). This performance may be short of what is expected from a vulnerability detection model, and software developers may remain skeptical about using a tool that only tells that a vulnerability is present, without giving more details of where to look for it, adding to the fact that about one-third of the times it may be wrong.

In this scenario, interpretability methods can be of great help, enhancing the machine learning model's capacity to communicate what factors were the most important for its prediction. The purpose of the interpretability result is to convince the user that the model took its decision in the right way. Concretely, we will use interpretability methods to identify what parts of the input to the model were most relevant to its prediction, facilitating the developer's job of finding the precise location of the vulnerability in the code in order to fix it.

Several works have been done in the field of interpretability and explainability of the decision-making of Artificial Intelligence (AI) models and their inner workings (BURKART; HUBER, 2021). The surge in the number of studies in this area is much driven by the need and willingness to apply these models in ethically or socially sensitive tasks (e.g., health, justice), which has spurred even the creation of new legislation about the necessary transparency of machine learning models.

In the field of Natural Language Processing (NLP), several methods of interpretability were evaluated for the task of text classification (ATANASOVA *et al.*, 2020). It was found that gradient-based methods, known as saliency methods, (SUNDARARAJAN *et al.*, 2017) perform best across model architectures, and for the Transformer architecture in particular, InputXGradient gave the best results (KINDERMANS *et al.*, 2016). The evaluation of interpretability methods used several criteria, such as Agreement with Human Rationale and Rationale Consistency (ATANASOVA *et al.*, 2020), which were only possible to be calculated due to the availability of a suitable benchmark dataset, containing human annotation of the salient tokens, or the tokens a human would consider most important to make a decision.

For the task of software vulnerability detection, however, the use of interpretability methods to evaluate the models' prediction and decision-making has been more limited, and the assessment of these methods less objective.

We find that there is room for a more thorough evaluation of interpretability methods

for this task, similar to what has been done in Natural Language Processing. In particular, analysis of which ones of the proposed deep learning models have the ability to give a more clear explanation of their decision-making process would be useful for their adoption by developers. The main roadblock in this direction is, from our point of view, the lack of an accepted benchmark for the assessment of the interpretability methods results.

1.1 Objective

Our main objective is to provide a benchmark dataset and an objective process for the assessment of interpretability methods of machine learning models in the task of vulnerability detection. Secondly, we evaluate state-of-the-art language models using the two most accepted interpretability methods for the text classification task and evaluate their results, comparing their outputs and discussing the reason for the different performances.

1.2 Motivation

Questions about deep learning models' transparency and interpretability, such as which biases the models can present when making a decision, or what decision-making processes these models undergo, have been the center of discussion for years. These topics have been much debated in the fields of Natural Language Processing and Computer Vision, but they are a valid discussion for any machine learning model trained to make a decision in place of a human being or in its assistance.

A younger field being explored by machine learning is Programming Language (PL) understanding. Many of its subtasks, like code auto-completion, code generation, programming language translation and semantic search, have lately been subject to the scrutiny of machine learning models (LU *et al.*, 2021).

Also recently, several disruptions of critical systems were caused due to the malicious activities of attackers aiming at breaking down applications or accessing sensible data. These attacks are possible by exploiting security vulnerabilities in these systems, which usually originate in subtle errors made by their programmers. This becomes even more critical due to the widespread use of open-source libraries, where a vulnerability can be unknowingly propagated to many seemingly unrelated systems (RUSSELL *et al.*, 2018).

The field of vulnerability detection is traditionally based on static code analyzers, dynamic and symbolic testers. These traditional approaches present several drawbacks, including many false positive alarms, limited coverage of vulnerabilities or are too computationally expensive to run on large programs, preventing scaling the tests (BURATTI

et al., 2020; RUSSELL *et al.*, 2018).

Therefore, attention has been given to the application of machine learning models to this task. The approaches used have mainly taken two paths: the first tries to take advantage of the topology and structure of the code explicitly (ZHOU *et al.*, 2019; NGUYEN *et al.*, 2021), and the second exploits the hypothesis of code naturalness (HINDLE *et al.*, 2012; BURATTI *et al.*, 2020), which argues that Programming Languages can be understood using the same methods applied in Natural Languages, letting the models implicitly capture the language syntax and structure during training (RUSSELL *et al.*, 2018; FENG *et al.*, 2020; PHAN *et al.*, 2021).

Even though relative success is being achieved, especially by large pre-trained language models (FENG *et al.*, 2020; PHAN *et al.*, 2021), there is still a long road ahead, once the current state of the art still falls short of 70% accuracy in the most accepted benchmark (LU *et al.*, 2021).

Furthermore, the state-of-the-art models are trained to classify a function as vulnerable or not, which can be viewed as a rather coarse result and a limitation of the current methods, once this information is of limited use for the software developers. A more helpful outcome would be to highlight what are the most salient or important parts of the code for predicting it as vulnerable, and if possible, where exactly the vulnerability is present.

This more fine-grained result could be achieved by the use of interpretability methods, such as the ones being discussed for years in the fields of Computer Vision and Natural Language Processing (SUNDARARAJAN *et al.*, 2017; KINDERMANS *et al.*, 2017). The use of these methods has already made shy inroads in the field of vulnerability detection (SOTGIU *et al.*, 2022; RENIERES; REISS, 2003), but the lack of accepted benchmarks to evaluate and compare these methods makes the research less objective than expected and prevents further advances.

Therefore, there is a need for the introduction of an accepted benchmark for interpretability methods for the task of vulnerability detection, as well as a clear way to compare and measure the preciseness of the interpretability results of different machine learning models and the quality of the output of different interpretability techniques.

1.3 Outline

In the remaining chapters, we present the literature background, the methods used in this work, and our results and contributions to the field of vulnerability detection in programming languages.

In Chapter 2, we review the literature on the subject, encompassing traditional approaches for vulnerability detection, machine learning for classification, the concept of deep learning and its application for vulnerability detection, the use of large pre-trained models, and finally, interpretability methods used to try and understand the inner workings of deep learning models.

In Chapter 3, we present the Methods used in this work, namely the choice of the pre-trained language models CoTexT and CodeBERT, the fine-tuning process for the task of vulnerability detection, the implementation of the interpretability methods Saliency and InputXGradient, the interpretability benchmark dataset curation, and the choice of evaluation metric.

In Chapter 4, we present the fine-tuning and interpretability results obtained and provide a brief discussion, including a qualitative comparison of the interpretability results achieved by both interpretability methods as well as by both language models.

Finally, in Chapter 5, we conclude our work by observing that, as in Natural Language Processing, InputXGradient is the interpretability method capable of generating the best outputs, and that the CodeBERT model, despite presenting a slightly inferior vulnerability detection accuracy compared with CoTexT, is superior to the latter by a considerable margin when comparing the capacity of the models to render an interpretation of their predictions. We also highlight the effects the model tokenizer can have on its ability to learn the semantic and syntactic structure of the programming language, as well as on its capacity to generate an interpretable output.

2 Fundamental Background

In this chapter, we present a literature review covering traditional approaches for vulnerability detection, machine learning for classification, the concept of deep learning and deep learning methods applied for the task of vulnerability detection, pre-trained language models, and interpretability methods for deep learning models.

2.1 Traditional Approaches for Vulnerability Detection

The simplest approach for vulnerability detection is the application of static analyzers. The analysis is done without the need to compile or execute the code, but by the identification of code structures representing flaws. The identification process is generally based on handcrafted rules or pattern matching the code being analyzed with a database of common vulnerabilities (WHEELER, 2018; INC., 2013).

The main weakness of static analyzers is that they usually generate a high number of false positive alarms, making it counter-productive for developers to check all of them and hindering the use of these tools in practice (CHEIRDARI; KARABATIS, 2018).

A second approach is dynamic analysis, which repeatedly executes the program in real or virtual processors with different test inputs, ideally covering all possible outputs. For example, one such method compares the faulty runs with their closest neighbors which ran successfully, and based on this comparison highlights the suspicious parts of the code. Techniques similar to this provide filtering of possible code vulnerabilities, but do not provide support for explaining the cause of the error (RENIERES; REISS, 2003).

Finally, symbolic analysis probes all possible program paths by replacing the input data with symbolic values, and analyzing their use over the control flow graph of the program. The main drawback of this branch of techniques is that they are expensive to run, and may be impractical for large programs, once the number of feasible paths that need to be tested grows exponentially with the size of the program (KING, 1976; RUSSELL *et al.*, 2018).

2.2 Machine Learning for Classification

Machine learning models can be described as programs that have the ability to improve their performance in a given task as they gain experience on it. These models employ the induction principle, which is the process of inferring general rules from specific data: given a set of examples of a task, the models are capable of inferring a function or hypothesis to solve that task (FACELI *et al.*, 2011).

The kind of learning employed by these models can be divided into two macro-categories: supervised learning and unsupervised learning. Furthermore, supervised learning is also described as predictive, while the latter as descriptive. In descriptive learning, the data being presented to the model does not have an output to be predicted, so the goal of the model is to explore and learn how to describe this data. For instance, the model may learn what are the groups of objects in the data which are most similar, or what are the rules of association present in the data (FACELI *et al.*, 2011).

In contrast, in supervised learning it can be said that the model has a supervisor that knows the true output for any given input, and the job of the model is to learn the mapping between input and output. Supervised learning models are called predictors, as they should learn how to predict an output given an input (FACELI *et al.*, 2011).

Depending on the nature of the output, the prediction problem receives different names: if the output of the model is a real number, it is a regression problem, and if the output of the model is a discrete value representing a finite number of categories, it is a classification problem (FACELI *et al.*, 2011).

For the task of vulnerability detection, the model must predict if a snippet of programming language code is vulnerable or not. Therefore, it is a classification problem with two possible categories as outputs.

2.3 Deep Learning

Deep learning is a subset of machine learning, which uses a nested hierarchical structure to build complex representations based on simpler ones. For instance, a deep learning model can learn to identify complex features such as faces, by first identifying corners, lines, textures, and colors. Each of these features is identified by different functions which compose the model and are placed in a layered fashion, with the representation learned from one function being fed into the next (GOODFELLOW *et al.*, 2016).

In other terms, deep learning models are constructed by the stacking of simple functions. This stacking gives the model its depth, and it is the combination of these functions

that enables the model to learn abstract concepts. The canonical example is the multilayer perceptron (MLP), which is the sequential combination of simple mathematical functions, with the output of one function serving as the input to the next one. Accordingly, one possible interpretation of deep learning models is that each mathematical function gives as output a new representation of the input data, and as the network gets deeper, the richer and more abstract this representation becomes (GOODFELLOW *et al.*, 2016).

Deep learning methods were able to achieve great success in tasks that are subjective and intuitive, such as understanding speech and images. It is difficult to hard code rules and formal statements to solve these tasks, and deep learning handles these problems by extracting patterns from the raw data in an end-to-end fashion, learning rough representations from the data in the initial layers of the networks, and building on it to learn more sophisticated representations in the final layers. This ability makes deep learning models very useful in tasks involving complex input data (GOODFELLOW *et al.*, 2016).

2.4 Deep Learning for Vulnerability Detection

In this section, we explore the main deep learning approaches in the literature for the task of vulnerability detection, which can be divided into early works exploring the naturalness hypothesis, graph-based approaches, and large pre-trained language models.

2.4.1 The Naturalness of Programming Languages

The first studies applying deep learning for vulnerability detection leveraged the naturalness hypothesis (BURATTI *et al.*, 2020; HINDLE *et al.*, 2012), which argues that programming languages can be understood and dealt with using the same tools which are applied to natural language.

Russell *et al.* (2018) started this trend, building on Natural Language Processing (NLP) techniques. Additionally, the authors released the first full-fledged large dataset for vulnerability detection in source code, composed of functions labeled as vulnerable or not, with more than 1 million examples.

The method proposed by Russell *et al.* (2018) is much similar to the approach first presented for sentence classification tasks, such as sentiment analysis and question classification (KIM, 2014). Firstly, the large corpora of functions is divided into tokens, which collectively form the vocabulary. Each token has its meaning represented by a dense embedding vector, which is learned during the network training.

The input to the network are functions to be classified as vulnerable or not. Each function is divided into its tokens, with each token represented by its embedding vector.

The concatenation of the input embedding vectors form an embedding matrix, which is the first layer of the neural network.

The embedding matrix is fed into a 1-dimensional Convolutional Neural Network (CNN) layer, which is followed by a pooling layer, a multilayer perceptron (MLP), and finally ending in a sigmoid non-linearity, which gives the probability of the function being vulnerable or not. The overall architecture can be seen in Figure 2.1.

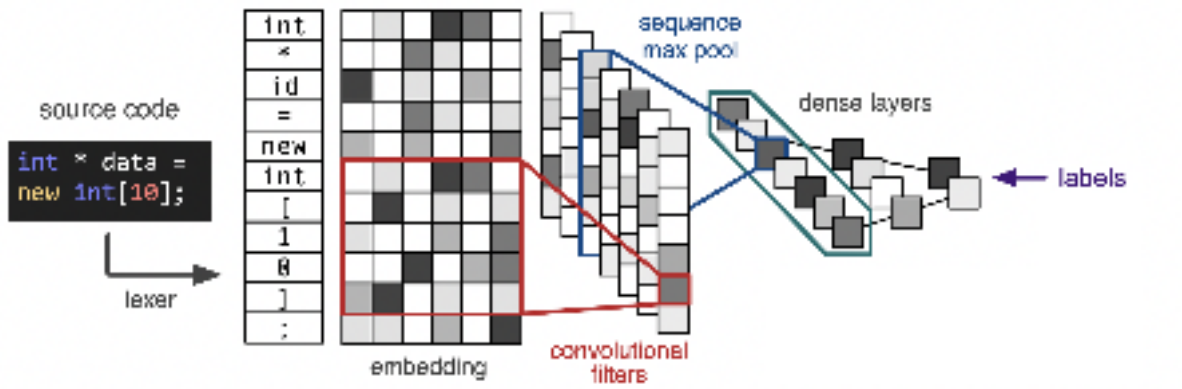


FIGURE 2.1 – Illustration of the architecture introduced by Russell *et al.* (2018) for vulnerability detection. Image from Russell *et al.* (2018).

Later, other works using the naturalness hypothesis followed, including Dam *et al.* (2021), which proposed applying Recurrent Neural Network LSTM-based model to detect vulnerabilities in Android applications.

2.4.2 Graph Neural Networks

With the advancements in the field of Graph Neural Networks (GNN), subsequent works on vulnerability detection tried to use this architecture style to take advantage of the structure and logic of programming languages, seeing the naturalness hypothesis as a weakness of the previous methods (ZHOU *et al.*, 2019).

The first work in this front (ZHOU *et al.*, 2019) mapped each code function into a heterogeneous graph, formed by the combination of the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG) and the natural code sequence (NCS). An example of one such graph can be seen in Figure 2.2. The graph is formed by four types of edges, derived from the four individual graphs, and share the same set of nodes. Each node, in turn, contains two attributes, its source code and type, which are represented by an embedding vector. The graph is fed into a graph neural network to learn its nodes' latent embeddings. These embeddings are then combined to form the graph latent representation by the use of a multilayer perceptron (MLP), which is followed by a sigmoid non-linearity, giving the probability of the function being vulnerable or not.

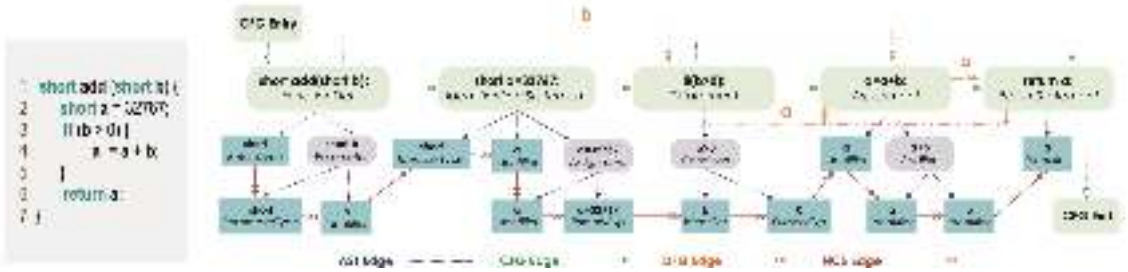


FIGURE 2.2 – Illustration of a heterogeneous graph used by the model introduced from Zhou *et al.* (2019). In the left is presented the code snipped from which the graph originated, and in the right the resulting graph representation. Image from Zhou *et al.* (2019).

Nguyen *et al.* (2021) also explored the idea of using graph neural networks to classify code functions, but introduced a simpler and more practical approach for inferring the graph from the source code. The method proposed processes the source code as a flat sequence of tokens, and presents two alternatives for graph inference:

- Unique token-focused construction: Unique tokens are represented by nodes in the graph, and two nodes are connected by an edge if they co-occur in the code within a fixed-size sliding window (Figure 2.3 Top);
- Index-focused construction: All the tokens are represented as nodes, and again two nodes are connected by an edge if they co-occur in a fixed sliding window (Figure 2.3 Bottom).

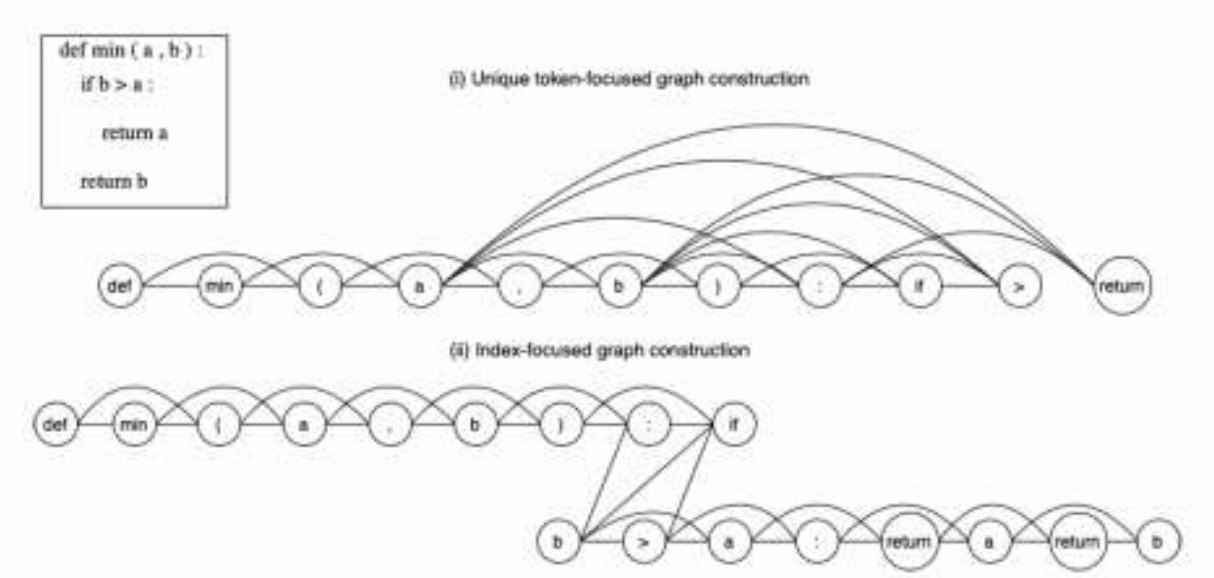


FIGURE 2.3 – Methods presented by Nguyen *et al.* (2021) for graph inference. Top: Unique token-focused graph construction. Bottom: Index-focused graph construction. Image from Nguyen *et al.* (2021).

The node embeddings initialization is done using the representation learned by a large pre-trained language model (FENG *et al.*, 2020).

The deep learning model proposed by Nguyen *et al.* (2021) is illustrated at Figure 2.4. In the first part, it employs a two-layer Graph Neural Network (GNN) with residual connections. The latent graph resulting from the first part is passed through a ReadOut Layer, which applies a mixture of sum pooling and max pooling layers to the node embeddings to calculate an aggregated graph representation. The graph representation is then passed to a sigmoid layer, giving the final prediction.

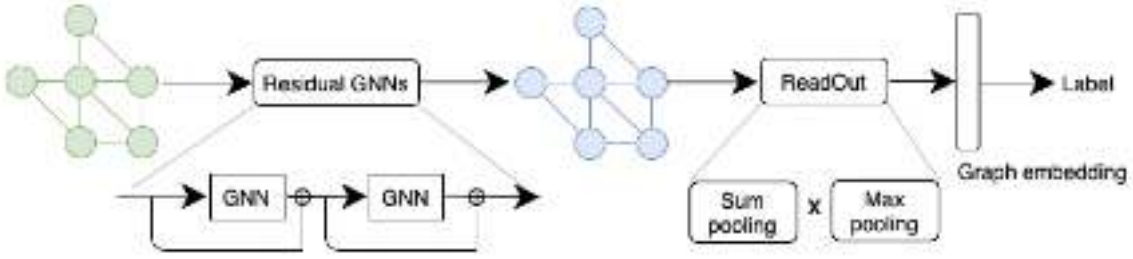


FIGURE 2.4 – Architecture of the model proposed by Nguyen *et al.* (2021). Image from Nguyen *et al.* (2021).

2.4.3 Pre-Trained Language Models

The pendulum shifted again to Natural Language Processing techniques with the introduction of the Transformer architecture (VASWANI *et al.*, 2017), especially with the use of large pre-trained language models fine-tuned for specific tasks (DEVLIN *et al.*, 2018; LIU *et al.*, 2019b; RAFFEL *et al.*, 2019). In this section, the concepts and motivation of the Transformer architecture and pre-trained language models are reviewed, and its applications for vulnerability detection are explored.

2.4.3.1 Transformer Architecture

Encoder-decoder architectures employing Recurrent Neural Networks (RNN) have been the state-of-the-art approach for language modeling in Natural Language Processing tasks, such as machine translation and text summarization. However, a fundamental constraint of this architecture is its sequential nature, which has two main disadvantages (VASWANI *et al.*, 2017):

- Bottleneck effect: When two words are far apart in a sequence, it is hard for the model to learn dependencies between them, due to the vanishing gradients problem and limited memory, even for RNNs using more sophisticated units such as LSTM and GRU.
- Precludes parallelization in the time dimension: It prevents parallelization within training examples, once the calculation of the hidden state of a later token is de-

pendent on the hidden states of all the previous tokens, which have to be calculated first.

These weaknesses represent an upper limit for both the quality of the results obtained and for the practical use of this architecture in large models.

A solution for the first problem, which became an integral part of the RNN encoder-decoder architecture, is the use of attention mechanisms. Attention allows each step of the decoder to use a direct connection to the encoder, enabling it to focus on any particular part of the input sequence. This is accomplished by means of calculating attention coefficients for each one of the input tokens, and averaging their hidden states using these coefficients. This average of the input tokens hidden states is processed by the units of the decoder together with their own inputs, via concatenation or summation. In short, attention enables decoder units to attend to any word of the input when predicting a new word, regardless of their distance.

A more general definition of attention is that, given a set of vector values and a vector query, attention is a method to compute a weighted sum of the values, dependent on the query. The weighted sum is a selective summary of the information contained in the values, where the query determines on which values to focus on.

The limitations of the RNN architecture and the idea of attention led to the proposal of the Transformer architecture (VASWANI *et al.*, 2017), which entirely avoids the use of recurrence, replacing it with attention mechanisms. The substitution of recurrence by attention means that the operations in the time dimension can be parallelized, and consequently constant time complexity is achieved with respect to the sequence length. The Transformer model architecture is illustrated in Figure 2.5.

The proposed model architecture is composed of an encoder and a decoder. The attention mechanism of the encoder is called self-attention: each token on the input sequence pays attention to other tokens of the same sequence. In other terms, the query and value sentences are the same.

Additionally, the attention mechanism used in the Transformer model, besides using queries and values, utilizes keys as well, which are used to calculate the attention coefficients between the query and value sentences.

Let $x_i, ..., x_t$ be the embedding vectors of each token of the input sequence to the model, where t is the maximum length of the sequence, and $x_i \in \mathbb{R}^d$. Then, keys, queries,

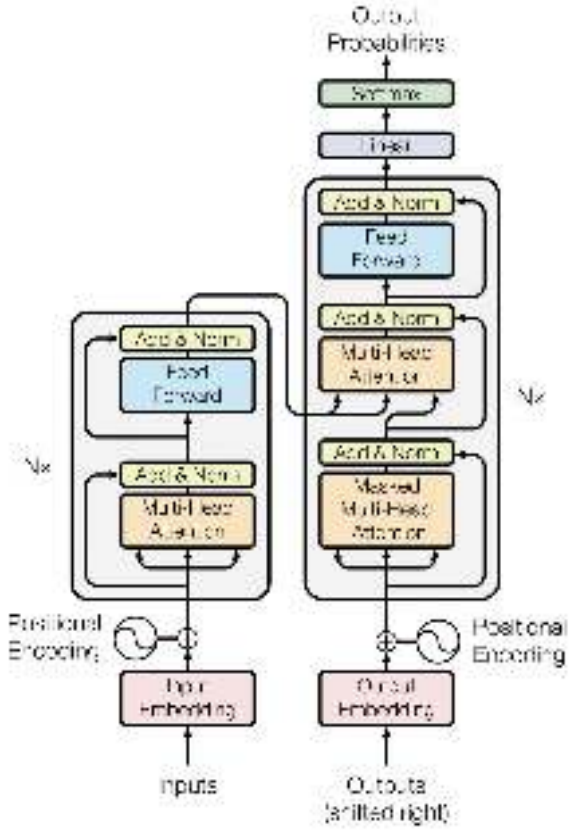


FIGURE 2.5 – The Transformer model architecture. Image from Vaswani *et al.* (2017).

and values vectors are:

$$\begin{aligned} \tilde{k}_i &= Kx_i \\ \tilde{q}_i &= Qx_i, \\ \tilde{v}_i &= Vx_i, \end{aligned} \tag{2.1}$$

where $K \in \mathbb{R}^{dxd}$, $Q \in \mathbb{R}^{dxd}$ and $V \in \mathbb{R}^{dxd}$ are the key, query and value matrices, respectively. These matrices are learned during the model training, and should allow different aspects of the x_i vector to be emphasized in each of the three roles.

Attention, however, cannot be used as a drop-in replacement for recurrence, once it is an operation in sets: it has no inherent notion of order, and in Natural Language Processing the ordering of words matter. To encode the notion of relative or absolute position of the tokens in the sequence, positional embeddings are added to the input token embeddings. Positional embeddings are vectors of the same size as the input embeddings, and they can be generated in two ways: they can be learned during the network training, or they can be fixed. In the original paper (VASWANI *et al.*, 2017), fixed positional encodings were used, being a concatenation of sinusoidal functions of varying periods.

Consider representing each sentence index as a vector $p_i \in \mathbb{R}^d$, for $i \in 1, 2, ..., t$. To

incorporate the sequence indexing to the queries, keys and values vectors, a simple vector summation suffices:

$$\begin{aligned} k_i &= \tilde{k}_i + p_i, \\ q_i &= \tilde{q}_i + p_i, \\ v_i &= \tilde{v}_i + p_i. \end{aligned} \tag{2.2}$$

After this, the attention weights are then computed in a matrix calculation which can be easily parallelized.

Let $X = [x_1; \ldots; x_t] \in \mathbb{R}^{txd}$ be the matrix containing the input vectors for all tokens in the input sequence, and K, Q and $V \in \mathbb{R}^{dxd}$ be the key, query and value matrices, respectively. The weighted output of the attention layer, which is called dot-product attention, is

$$\text{output} = \text{softmax}\big(XQ(XK)^T\big)\, XV, \tag{2.3}$$

where output $\in \mathbb{R}^{txd}$.

A drawback of the dot-product attention is that, when the dimensionality d becomes large, the dot products between vectors tend to become large too. This results in the softmax function outputting a probability distribution giving high scores to very few of its components, and making the gradient to everything else small. To overcome this, the attention mechanism is scaled by the square root of the dimensionality of its vectors, receiving the name *scaled dot-product attention*:

$$\text{output} = \text{softmax}\left(\frac{XQ(XK)^T}{\sqrt{d}}\right) XV, \tag{2.4}$$

where output $\in \mathbb{R}^{txd}$.

Another limitation of pure attention mechanisms is that they calculate only weighted average of vectors, having no non-linearities. To add non-linearities to the model, feed-forward networks with ReLU non-linearities are added for every individual token embedding after the attention layer:

$$m = W_2 \text{ReLU}(W_1 \text{output} + b_1) + b_2, \tag{2.5}$$

where W_1, W_2, b_1 and b_2 are learnable matrices and biases, and $m \in \mathbb{R}^{txd}$.

Two more changes to the basic attention mechanisms are the addition of residual connection and layer normalization. Firstly, residual connections are added for each component of the encoder and decoder because these connections are thought to make the loss landscape considerably smoother, thus making the network easier to train (LIU *et al.*,

2019a).

Secondly, after the sum of the residual connection, the result is normalized to unit mean and standard deviation. The reason for this operation is to cut down on uninformative variation on hidden values, which is argued to improve training due to its normalizing of gradients (XU *et al.*, 2019).

Finally, the decoder of the transformer architecture, besides having a self-attention layer, counts also with a cross-attention layer, in which its input sequence attends to the tokens of the output sequence from the encoder. In this case, the query vector is from the decoder sequence, while the keys and values are from the output of the encoder.

Another distinction is that the decoder of the model has the role of predicting a sequence, token-by-token. Therefore, it cannot have the ability to look at the future; that is to say, each token must have the ability to calculate attention coefficients only to the tokens already generated. To accomplish this without losing the benefit of parallelization, the future words are masked with attention scores of minus infinity.

2.4.3.2 Pre-Trained Language Models

The use of pre-training language models started with the pre-training of word embeddings, with methods like Word2Vec and GloVe (MIKOLOV *et al.*, 2013; PENNINGTON *et al.*, 2014). In Word2Vec, for instance, the dense vector representing each word in the vocabulary is learned by predicting the probability of the context words given a center word. Therefore, the trained word embeddings incorporate information about the words that occur in the same context, placings words of similar meaning close to each other in the high-dimensional space in which they are embedded.

After pre-training, the resulting word embeddings can be deployed in the training of a generic machine learning model for any natural language processing task, like text classification or language modeling. This alleviates the need for the model to learn from scratch the meaning of all the words in the vocabulary during training: it is able to focus solely on learning how to perform its downstream task, instead of learning the language while at the same time having to learn how to perform its task.

Further advances in pre-training occurred after the introduction of the Transformer architecture: First, Radford e Narasimhan (2018) noted in their paper, which introduces the GPT model, that unlabeled natural language data could be gathered in large amounts, while the availability of labeled data for specific tasks was more restricted. As a result of this limitation in the quantity of labeled data, it was challenging to train discriminative machine learning models using only the training data available for the downstream task, once it had to be sufficient to teach all contextual aspects of the language to the model.

Radford e Narasimhan (2018) argued that models which are able to learn good language representations from unlabeled data should be able to achieve an important boost in performance. In their paper, they provided a first approach to pre-training the entire model with a large amount of unlabeled data, and fine-tuning all pre-training parameters for the downstream task using labeled data, with minimal change to the model architecture. The GPT model introduced by them was a Transformer decoder-only, and it used language modeling as the pre-training task.

Later, the introduction of the BERT model (DEVLIN *et al.*, 2018) gave further momentum to the pre-training paradigm by making the following change: pre-training a Transformer encoder-only model, instead of a decoder-only as in the GPT case. The reasoning behind the architecture choice was that an encoder can use bidirectional context via self-attention, which is not possible in the decoder, once it would allow the language model to attend to future words. The restriction of having only unidirectional context can be harmful in tasks where it was beneficial to incorporate the context in both directions, such as question answering.

Also, because it uses bidirectional attention, the BERT model cannot be pre-trained with standard language modeling tasks. Therefore, the authors proposed the masked language modeling (MLM) task, in which a fraction of the input words are hidden from the model, substituted by a *MASK* token, and the model task is to learn to predict these hidden words.

A third influential pre-trained language model is T5, which uses the complete Transformer architecture, comprising encoder and decoder (RAFFEL *et al.*, 2019). The T5 authors also proposed the Span Corruption pre-training task as the pre-training objective for the model. In Span Corruption, different-length spans from the input are replaced by unique placeholders, and the model should learn to decode out the spans that were removed from the input.

The pre-training paradigm has shown impressive results, and has become the go-to approach for Natural Language Processing tasks. The reason pre-trained models work so well appears to be that the local minimum found during stochastic gradient descent at the fine-tuning stage remains relatively close to the local minima found in the pre-training stage. Additionally, this local minimum usually is found to generalize well, along with the fact that the gradients of the fine-tuning loss near this region propagate nicely (ERHAN *et al.*, 2010).

2.4.3.3 Pre-Trained Language Models for Programming Languages

Feng *et al.* (2020) were the first to build a large natural language (NL) and programming language (PL) multi-modal pre-trained language model, called CodeBERT. To train

the model, they used both unimodal data (only NL or PL data), and multi-modal data (PL data paired with their documentation in NL). In particular, the model was trained in code of 6 different programming languages. In the pre-training stage, the model went through two tasks: masked-language modeling and replaced token detection, where the model is trained to detect whether a word is the original or not. After pre-training, CodeBERT can be fine-tuned for a variety of code-understanding tasks, like natural language code search and vulnerability detection.

Several others multi-modal pre-trained models were released subsequently (HANIF; MAFFEIS, 2022; CHEN *et al.*, 2021; AHMAD *et al.*, 2021; PHAN *et al.*, 2021). In particular, Phan *et al.* (2021) introduced the CoTexT model, a pre-trained language model using the T5 architecture (RAFFEL *et al.*, 2019). As in the case of CodeBERT, the model can be fine-tuned for a range of code understanding tasks, and achieved state-of-the-art results in the CodeXGlue (LU *et al.*, 2021) competition for vulnerability detection, which is the main benchmark for the task.

2.5 Interpretability Methods

In this section, we introduce the prolific field of interpretability for deep learning models, focusing on their applications in Natural Language Processing, which is the main test bed for these methods. We further explore the branch of interpretability most relevant to our research.

2.5.1 Interpretability Methods for Deep Learning Models

The problem interpretability methods try to solve is how to attribute the prediction of a machine learning model to its input features (SUNDARARAJAN *et al.*, 2017), such that one may have knowledge of what drove the model decision-making. This understandability is critical for the practical use of these models in tasks that involve ethical issues or life-critical decisions, such as judicial sentencing and medical diagnosis (SUNDARARAJAN *et al.*, 2017).

One challenge in the design and evaluation of interpretability methods is that it may be hard to assess how well the method does empirically, and also, it may be hard to disentangle errors stemming from the misbehavior of the machine learning model versus the misbehavior of the interpretability method (SUNDARARAJAN *et al.*, 2017).

While there is a growing quantity of interpretability methods available, it is found that they may produce varying, and sometimes contradicting answers to the interpretability question (ATANASOVA *et al.*, 2020). This makes it important to be able to assess such

methods, in order to choose the most appropriate one for the machine learning model architecture in question and its application. This assessment may be further complicated by the lack of interpretability ground truth, therefore benchmarks where the salient or most important input features for the given output are known are needed for an objective evaluation of the interpretability techniques (KINDERMANS *et al.*, 2017).

Interpretability methods can be categorized by their approach to explain the machine learning model prediction: employment of the model gradients, perturbation-based, or providing explanation through model simplifications (ATANASOVA *et al.*, 2020). Gradient or saliency-based approaches compute the gradient with respect to the input of the model to quantify the importance of the input features (SIMONYAN *et al.*, 2013). Variations of this basic method are InputXGradient, which multiplies the gradient with the input (KINDERMANS *et al.*, 2017), and Guided Backpropagation, which overwrites the gradients of the ReLU functions so that only non-negative gradients are backpropagated (SPRINGENBERG *et al.*, 2014). Perturbation-based methods include Occlusion, where each token is replaced by a baseline token, measuring the change in output (ZEILER; FERGUS, 2013), and Shapley Value Sampling, in which the average marginal contribution of each input feature is approximated (CASTRO *et al.*, 2009). Simplification-based methods include LIME, in which the local decision boundary is approximated by a linear model (RIBEIRO *et al.*, 2016).

Atanasova *et al.* (2020) performed a large assessment and comparison between interpretability methods applied to Natural Language Processing tasks, evaluating the methods applied to different machine learning model architectures (Transformers, Convolutional Neural Networks, and Recurrent Neural Networks) and in three different benchmark datasets. The authors found that gradient-based interpretability methods gave the best results across all architectures and in all benchmark datasets. In particular, Atanasova *et al.* (2020) showed that for the Transformer architecture, InputXGradient was the best-performing interpretability technique overall.

2.5.1.1 Gradient Based Approaches

Simonyan *et al.* (2013) first proposed the use of saliency maps to find the most important pixels in an image for the task of image classification. Considering a linear classifier, and given a vectorized (one-dimension) image I and a class score function $S_c(I)$, we have that

$$S_c(I) = w_c^T I + b_c \ , \tag{2.6}$$

where w_c and b_c are the weight matrix and bias vector, respectively.

Taking the derivative of S_c with respect to the image, we have that

$$w_c = \frac{\partial S_c}{\partial I}. \tag{2.7}$$

One interpretation for this equation is that the magnitude of the derivative indicates which pixels need to be changed the least to affect the score of the classification task the most (SIMONYAN *et al.*, 2013). Therefore, these pixels should be the most important or most salient for classification, and this should generalize to deep neural networks.

This basic idea gave origin to the Saliency method of interpretability. A subsequent improvement over this technique, named InputXGradient (KINDERMANS *et al.*, 2017), multiplies the gradient backpropagated to the input layer with the weights of the input layer itself. This change was proposed to mitigate the effect noise in the input had on the resulting saliency map.

In the case of Natural Language Processing, the saliency map is calculated by backpropagating from the output token to the embedding layer. This operation results in a gradient vector of the same dimension as the embedding vector for each of the input tokens. To arrive at a saliency score for each token, the vectors need to be aggregated, and Atanasova *et al.* (2020) found that the Euclidean norm was the aggregation form that gave the best results when comparing to alternatives, such as taking the mean of the vector.

2.5.2 Interpretability in Vulnerability Detection

Studies applying interpretability methods in the field of code understanding, and in particular vulnerability detection, are few and far between.

Sotgiu *et al.* (2022) analyzed the CodeBERT model fine-tuned for vulnerability detection, by means of calculating the Shapley values of the input features in order to draw their importance, with the objective of discovering what were the most important tokens responsible for positive and negative classifications. The authors found that special characters of the programming language (e.g., {, }, $*$) were the most important tokens to classify both classes (vulnerable and not vulnerable). Also, they found that the model tended to classify a sample as vulnerable if it differed from the average distribution of the samples seen in training (for instance, if the sample had out-of-vocabulary tokens). Even though this study hints at the inner workings of the model assessed, the analysis of the importance of 1-grams does not provide the framework necessary to identify what drove the model to its decision, nor does it help narrowing down what are the parts of the code which are vulnerable.

Wattanakriengkrai *et al.* (2022) used the simplification-based LIME technique to find defect-prone lines in the code. The authors used a simple logistic regression classifier over a bag of tokens to classify functions as vulnerable or not. Despite the use of the interpretability technique, the framework proposed by the authors took some problematic shortcuts, like the removal of non-alpha-numeric characters during the data preprocessing phase, which may result in losing fundamental code structure. Additionally, in their study, they converted programming language code to bag-of-tokens, which considers only the frequency of the tokens in the code, completely ignoring word order. This conversion may render it impossible for any machine learning model to learn where in the code the vulnerability is present.

Mosolygó *et al.* (2022) took an altogether different approach: instead of predicting if a function is vulnerable or not, the authors proposed evaluating the code line-by-line to predict if it presented a vulnerability. In their work, the authors used ground-truth code lines with known vulnerabilities to build vector representations of these vulnerabilities using the word2vec embedding technique (MIKOLOV *et al.*, 2013). Subsequently, each line of the code being evaluated was embedded in a vector using the same technique, and compared with the representation of the known vulnerabilities using cosine distance. This approach is simple enough, but using single-line detections may not be sufficiently robust for vulnerability detection in real-world applications, once a vulnerability may be the result of the construction of several lines of code, or the interaction between different components. Additionally, the authors did not provide a comparison of their classification results against state-of-the-art techniques, such as CodeBERT or CoTexT (FENG *et al.*, 2020; PHAN *et al.*, 2021).

Finally, Wan *et al.* (2022) studied the pre-trained language models CodeBERT and GraphCodeBERT to understand their inner workings, investigating why these models worked and what features they had learned from code. Their analysis revealed that the self-attention mechanism of the Transformer encoder can capture the motif structure of the Abstract Syntax Tree, and that the hidden embedding vectors learned by the model also capture the syntax structure of the code. The ability of pre-trained models to learn the syntactic structure of the code helps to explain why these models are able to give superior results compared with Graph Neural Networks, which are trained explicitly using graph representations of the code, in order to leverage its structure.

3 Methods

In this section, we first present the pre-trained models used in this study and their fine-tuning procedure, next we describe the implementation of the interpretability methods chosen, and finally, we go through the curation procedure to generate a benchmark dataset for the interpretability results, as well its evaluation metric.

3.1 Pre-trained Models and Fine-Tuning

We chose two bimodal pre-trained language models for our studies: CodeBERT (FENG *et al.*, 2020) and CoTexT (PHAN *et al.*, 2021). Both are pre-trained in natural language and programming languages. We chose these models because CodeBERT is currently the most studied pre-trained model for programming languages in the literature, being present in most comparisons, and CoTexT is the current state of the art for the task of vulnerability detection in the benchmark competition CodeXGlue (LU *et al.*, 2021).

Both models are available in their pre-trained format together with their tokenizers in the *Hugging Face* library. Therefore, fine-tuning them for the vulnerability detection task is necessary.

The models are fine-tuned using the training partition of the Devign dataset (ZHOU *et al.*, 2019), which comprises approximately 20 thousand C/C++ functions, hand-labeled by specialists as vulnerable or not vulnerable, with balanced classes.

The fine-tuning process is slightly different for both models once CodeBERT is a Transformer encoder-only-based model, and CoTexT is a Transformer encoder-decoder-based model. For the first, we train a prediction head from scratch on top of the encoder model. This prediction head is simply a one-layer perceptron followed by a sigmoid activation layer. For the latter, in contrast, the decoder is a language model which generates tokens auto-regressively. Therefore, we fine-tune it to generate as the first token the strings *0* or *1*, which are the labels of our classification problem, effectively converting it to a binary classifier. The fine-tuning architectures for both models are illustrated in Figure 3.1. To avoid overfitting, we used early stopping with patience during the optimization process.

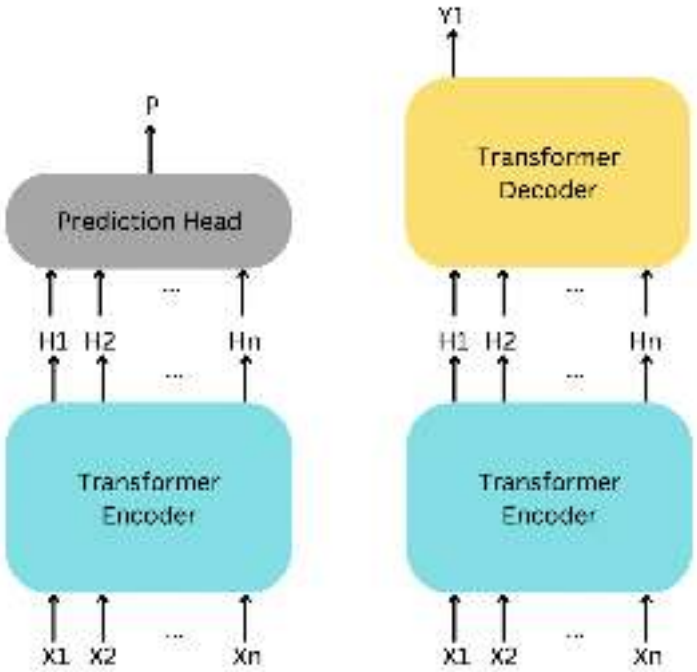


FIGURE 3.1 – Left: Architecture for the CodeBERT fine-tuning. We place a prediction head on top of the CodeBERT model, which is a Transformer encoder. The prediction head is composed of a single perceptron layer and a sigmoid activation function, which outputs the probability of the input function being vulnerable or not. Right: Architecture for the CoTexT model fine-tuning. The model is based on the T5 architecture, which is constituted by the complete transformer architecture, with both encoder and decoder. The decoder is a language model, and we fine-tune it to generate as its first output token the strings *0* or *1*, which are the labels of our classification problem.

3.2 Interpretability Methods

For the interpretability method, we chose two techniques: the InputXGradient method, shown by Atanasova *et al.* (2020) to be the best-performing interpretability technique for Natural Language Processing tasks using the Transformer architecture, and the original Saliency method, to be our baseline for comparison.

The process of implementation of both models is straightforward: After predicting a label for a given code function, we backpropagate the gradients of the network to the embedding layer. For the original Saliency method, the resulting gradient matrix is the saliency matrix, holding the importance vector for all tokens in the vocabulary. For the InputXGradient, the matrix of gradients is subsequently multiplied element-wise with the embedding matrix. The result of this operation is the saliency matrix used by the method.

We filter the saliency matrices to keep only the saliency vectors for the tokens present in the input sequence, and we aggregate each one of these vectors to arrive at a saliency or importance score for each input token. The saliency score represents the importance of that input token for the network prediction.

The two most common methods for aggregating the saliency vectors are calculating

the mean of each vector and its Euclidean norm. Atanasova *et al.* (2020) has shown that for the InputXGradient method, the Euclidean norm resulted in the best interpretability results. A possible limitation of the Euclidean norm aggregation is that it completely loses the information on the direction of the vector, remaining only with the information of its magnitude. Therefore, tokens that have the same magnitude saliency vectors, but point to completely different directions, will be indistinguishable. Consequently, the method may not be able to identify or separate tokens that are pushing the prediction of the model to different classification labels.

To test this hypothesis, we compare the InpuXGradient with Euclidean norm aggregation with the original Saliency method employing mean aggregation. The mean aggregation should do better at preserving information regarding the direction of the vector, although losing the magnitude information.

For both methods, the saliency score is an unbound number. In order to make it suitable to represent the importance score of the input tokens, as well as to be presented in a heatmap of saliencies, we make the following transformations:

- InputXGradient: The saliency scores aggregated by the Euclidean norm are unbounded positive numbers. We scale them to the range from 0 to 1, where 0 means the respective token does not have any importance for the model prediction, and 1 means the respective token has the most importance for it.
- Saliency: The saliency score of each token is divided by the standard deviation of the saliency scores of the input sequence, resulting in values between -3 and +3. This choice of scaling was preferred to preserve the mean of the distribution.

The pseudo-code for the InputXGradient method is:

```
gradientAtEmbeddings = model.embeddings.weight.grad
embeddingsTimesGradients = model.embeddings.weight *
    gradientAtEmbeddings
saliencyList = []
for tokenId in input_tokens:
    saliencyVector = embeddingsTimesGradients[token_id, :]
    saliencyScore = saliencyVector.norm(p=2)
    saliencyList.append(saliencyScore)
scaler = MinMaxScaler(feature_range(0, 1))
saliencyScores = scaler.fit_transform(saliencyList)
```

The pseudo-code for the Saliency method is:

```
gradientAtEmbeddings = model.embeddings.weight.grad
saliencyList = []
for tokenId in input_tokens:
    saliencyVector = gradientAtEmbeddings[token_id, :]
    saliencyScore = mean(saliencyVector)
    saliencyList.append(saliencyScore)
saliencyScores = saliencyList / std(saliencyList)
```

3.3 Interpretability Benchmark Dataset

The availability of an accepted benchmark is necessary if any field of research is to advance consistently, being the best example the ImageNet competition in the computer vision field (RUSSAKOVSKY *et al.*, 2014). In the area of Programming Language understanding, this role is being fulfilled by the CodeXGlue set of benchmarks datasets (LU *et al.*, 2021), comprehending tasks like vulnerability detection, code repair, code summarization and others.

Using interpretability methods to understand why the model made its decision, the objective is to assess what was the importance of each of the model inputs to arrive at its final prediction. To conduct an evaluation of the interpretability methods, it is necessary datasets annotated with the salient tokens, that is to say, with annotations of what the most important tokens are for the ground-truth output for any given example (ATANASOVA *et al.*, 2020). Annotated datasets for the evaluation of interpretability methods are relatively easy to obtain for Natural Language Processing tasks (ATANASOVA *et al.*, 2020), but in contrast, are lacking for Programming Language tasks such as vulnerability detection.

To fulfill this gap, we curated a benchmark dataset for interpretability methods in the vulnerability detection task. The dataset proposed presents the following characteristics:

- Each example in the dataset is a function written in the C/C++ programming language;
- Every function in the dataset contains at least one vulnerability;
- Each example in the dataset is accompanied by a binary mask the same length as the number of strings in the example: the mask has value 0 if that string is not part of a snippet of code which introduces a vulnerability, and has value 1 otherwise.

To develop this dataset, we leveraged the C/C++ code vulnerability dataset named Big-Vul (FAN *et al.*, 2020). Big-Vul contains 6,093 vulnerable functions collected from

Github, and all examples have the vulnerable snippet of code together with the patch used to fix the issue, accordingly annotated.

From that starting point, we removed the code patches, and from the resulting function we generated a mask with ones in place of the vulnerable snippet of code and zeros in the remaining code. An example from the dataset with its vulnerability highlighted is illustrated in Figure 3.2, and the complete dataset is publicly available (SILVEIRA *et al.*, 2023).

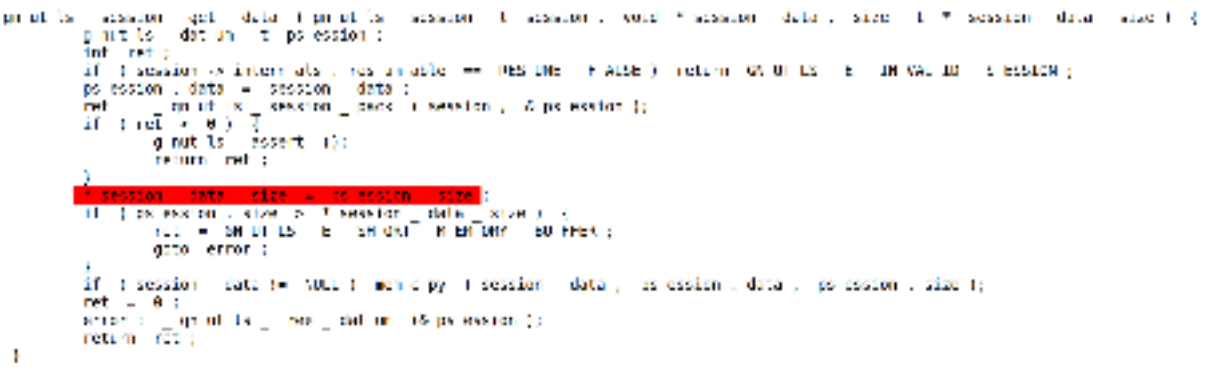


FIGURE 3.2 – Vulnerable sample from the dataset with the position of the vulnerability labeled. The function is tokenized, and each of its tokens is masked with values of 0 or 1. The collection of tokens masked as 1 represents the position of the vulnerabilities in the code, and is highlighted in red for illustration.

3.3.1 Evaluation Metric

To assess how well the heatmap resultant from the interpretability method matches the ground-truth mask, the following steps are taken:

1. Transform the heatmap of saliencies from the interpretability method into a binary heatmap. This is done using a threshold, where saliency/importance values above or equal to the threshold are rounded up to 1, and saliency/importance values below the threshold are rounded down to 0.

2. After the first step, the problem becomes a binary classification. For each example in the dataset, we calculate a precision score of how well the binary interpretability heatmap matches the ground truth mask.

3. With the precision score of each example in the dataset, we average the results, arriving at a Mean Precision Score for the whole dataset.

The Mean Precision Score is the final metric we use to evaluate how well the interpretability heatmap matches the true binary mask. We chose to measure precision

because it penalizes false positive detections, therefore benefiting predictive models with more sensitive results, and giving low marks to models that signal too many tokens as important. For this same reason we refrain from measuring recall, which penalizes false negative detections, and could benefit models that give high importance to too many tokens.

3.4 Interpretability Methods Comparison and Application for Different Language Models

We use the InputXGradient and the original Saliency method to interpret the outputs of our two fine-tuned models, CodeBERT and CoTexT. For each model, we predict the label for every function in the interpretability dataset. If the model predicts the function as vulnerable, we calculate and save the importance heatmap of its input.

Subsequently, we calculate the Mean Precision Score for each model, for thresholds varying from 0.1 to 0.9, with steps of 0.05, for the InputXGradient method, and for thresholds varying from 0.1 to 2.9, with the same step-size, for the Saliency method.

With the saliency heatmaps calculated, we qualitatively compare the heatmaps of both interpretability methods, InputXGradient and Saliency, and note the differences that set them apart, and which is the most appropriate for our setting, selecting the best one.

For the interpretability method chosen, we use the best threshold for each model, CodeBERT and CoTexT, and evaluate the results, observing how well the models lend themselves to interpretation, that is to say, how truthful their resultant saliency heatmaps are.

Finally, we discuss qualitatively the characteristics of each model and how they affect the capacity of the models in generating interpretability of their predictions.

4 Results

In this chapter we present the results of our experiments, along with brief discussions.

4.1 Fine-Tuning Results

After fine-tuning the CodeBERT and CoTexT models for the vulnerability prediction task using the Devign dataset training partition (ZHOU *et al.*, 2019), we evaluate them in the test partition, which is available at the CodeXGlue benchmark (LU *et al.*, 2021). The results are shown in Table 4.1.

TABLE 4.1 – Fine-tuning results for the models CodeBERT and CoTexT.

Model	**Accuracy**	**Precision**	**Recall**
CodeBERT	63.1%	63.1%	47.2%
CoTexT	64.4%	64.7%	49.3%

Both models show similar results, with CoTexT having a slight advantage. The result is in line with the CodeXGlue leaderboard, which presents CoTexT as the state-of-the-art model for the task of vulnerability detection.

A point worth noting is that the dataset used is composed only of C/C++ functions. Datasets for vulnerability detection in other programming languages, or even composed of multiple programming languages are lacking. Consequently, this creates a roadblock to the training and deployment of vulnerability detection models for other programming languages besides C/C++.

4.2 Interpretability Results

We apply the fine-tuned models to classify the samples of the interpretability benchmark dataset, composed uniquely of vulnerable functions. For the samples predicted as vulnerable by the models, we evaluate the resultant saliency heatmaps for the Saliency and InputXGradient methods against the ground-truth heatmaps.

Firstly, we report the results from the application of the InputXGradient method to the predictions of the CodeBERT and CoTexT models, which are shown in Table 4.2.

TABLE 4.2 – Mean precision achieved applying the InputXGradient method to the CodeBERT and CoTexT models.

Model	**Best Threshold**	**Mean Precision**
CodeBERT	0.10	31.3%
CoTexT	0.35	24.2%

The best results for the CodeBERT and CoTexT models were achieved with threshold values of 0.10 and 0.35, respectively. Both models achieved weak results, with CodeBERT presenting a mean precision score more than 7 percentage points above CoTexT.

Secondly, we evaluated the saliency heatmaps generated using the Saliency interpretability method, and the results are shown in Table 4.3. It can be observed that the results are very close to the ones from InputXGradient. It suggests that the Euclidean norm aggregation used in the InputXGradient, which loses the information on the direction of the saliency vector, is not hurting performance, or at least is as good as the mean aggregator employed by the Saliency method. Additionally, from the results, it seems that the Saliency and InputXGradient could be used interchangeably without considerable loss.

TABLE 4.3 – Mean precision achieved applying the Saliency method to the CodeBERT and CoTexT models.

Model	**Best Threshold**	**Mean Precision**
CodeBERT	0.10	31.1%
CoTexT	0.35	24.2%

To better understand the nature of the saliency scores generated by the InputXGradient and Saliency methods, observing if there is any preference to giving high or low scores to the tokens, we draw their histograms for both CodeBERT and CoTexT models. Figure 4.1 illustrates the histograms for the importance scores for the InputXGradient method, while Figure 4.2 illustrates the histograms for the importance scores for the Saliency method.

For the former, we can see that the distributions for the CodeBERT and CoTexT are similar, and represent a normal distribution truncated at zero. For the CoTexT model, it can be observed that the mean of the distribution is not centered at zero, but offset to the right. Additionally, both curves show a spike in the frequency of tokens with importance close to one, which can be interpreted to be the most important tokens as seen by the model.

For the Saliency scores histograms, the curves also represent normal distributions, and again are very similar for both models. We highlight that most tokens have importance

scores around the value of zero.

Additionally, when comparing the histograms for the two models, CoTexT and CodeBERT, it is noticeable the difference in the magnitude of their frequency: the scores for the CoTexT histograms have frequencies roughly twice the frequencies for the scores of the CodeBERT histograms. Once each importance score corresponds to a token appearance in the dataset, it can be concluded that the CoTexT tokenizer discretizes the dataset in a more granular manner, resulting in approximately twice as many tokens.

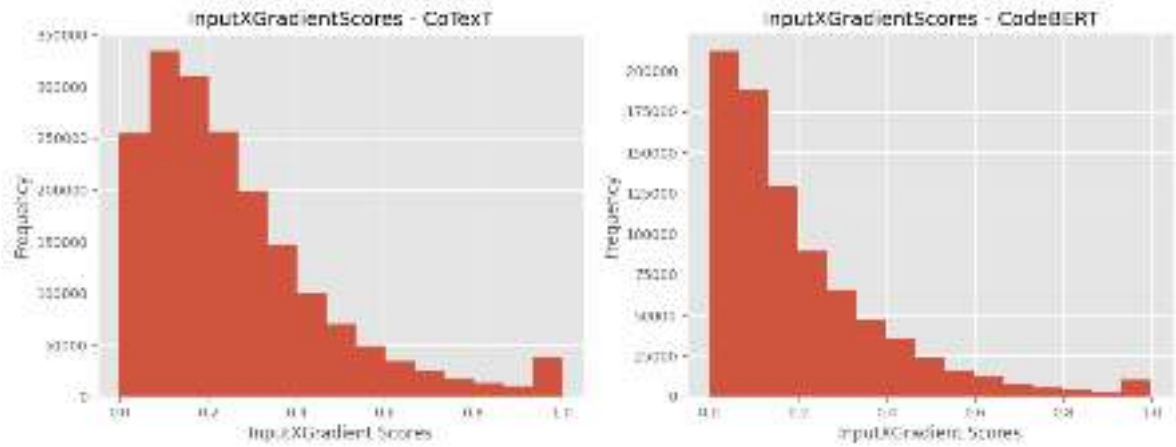


FIGURE 4.1 – Distribution of the InputXGradient importance scores for the CoTexT and CodeBERT models.

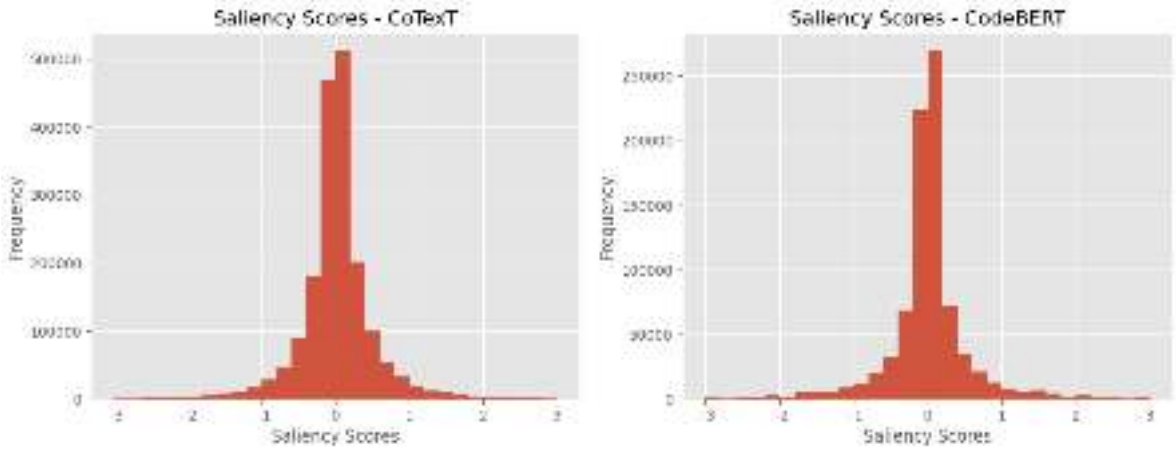


FIGURE 4.2 – Distribution of the Saliency importance scores for the CoTexT and CodeBERT models.

4.3 Qualitative Analysis of the Interpretability Results

In this section, we firstly compare the results of the InputXGradient and Saliency methods, and secondly we compare the interpretability heatmaps generated using the InputXGradient method for the CodeBERT and CoTexT models, discussing their differences and characteristics.

4.3.1 Comparison - InputXGradient and Saliency

Regarding the mean precision metric, both InputXGradient and Saliency methods achieved similar results. In this section, we compare qualitatively how interpretable are their generated heatmaps, and how well they are able to match the information present in the ground-truth. The heatmaps were generated using the interpretability methods applied to the CodeBERT model, which achieved the best mean precision score for both techniques.

Figure 4.3 shows a high-level overview of three vulnerable functions, one for each line of the illustration, and its heatmaps for the ground-truth annotation (left), InputXGradient (center), and Saliency (right) methods.

Firstly, we remember that for the ground-truth heatmaps (left), the exact location of the vulnerability is highlighted in red. Secondly, we note that the InputXGradient heatmaps (center) are also highlighted in red, with the most important tokens in a more opaque tone. Contrastingly, the Saliency heatmaps (right) are highlighted in both red and blue: the red color, as for InputXGradient, means a *positive* importance, and the blue color a *negative* importance. The reason for this is that the importance scores for the Saliency method range from -3 to +3, and we interpret that the more negative a token importance is, the more it influences the algorithm to predict that sample as *not vulnerable*, the opposite being true for positive importances.

Thirdly, when comparing the InputXGradient and Saliency heatmaps to the ground-truth tokens, it can be observed that the InputXGradient heatmaps offer a considerably cleaner picture, it being easier to identify the parts of the code considered as most important and to compare it with the ground-truth. For the saliency heatmaps, it is harder to grasp what are the parts of the code considered important and what are the ones that are not, once many parts of it are highlighted with both strong opaque tones of red and blue.

To help further support this point, Figure 4.4 shows the same heatmaps, but using a threshold to filter only the tokens with the highest importance. The heatmaps from the InputXGradient method (center column) are left with very few tokens as important, and we point out that the tokens highlighted are most often present in the positions where the vulnerabilities are. For the Saliency heatmaps, only tokens with positive importance are left, making it easier to observe where are the most important points. It is also observable that the Saliency method tends to give high importance to more tokens, with them sometimes being scattered over the code, making the task of finding the exact position of the vulnerability more difficult.

Therefore, once both methods achieved similar scores in the mean precision metric, and the InputXGradient heatmaps render an easier and more concise interpretation of its output, for the remaining sections we consider only the InputXGradient method.

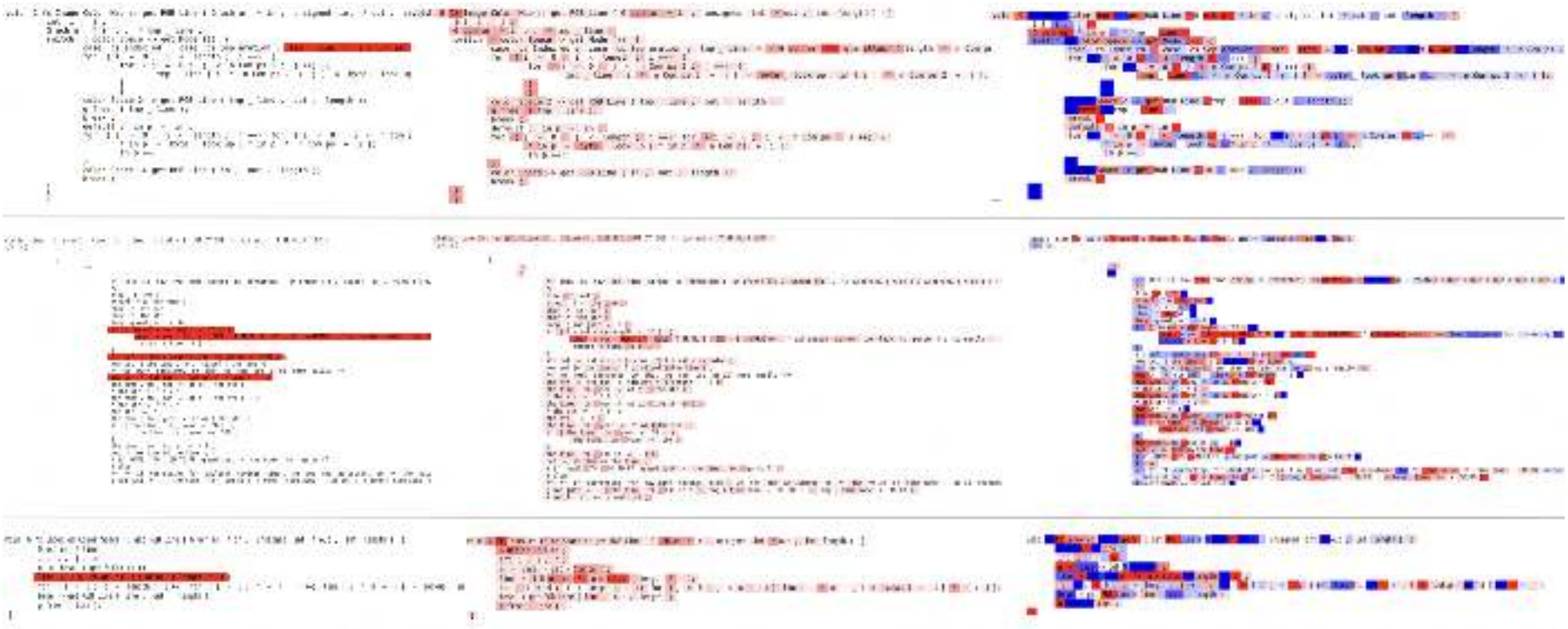


FIGURE 4.3 – Overview of the heatmaps of three vulnerable functions, represented each in one row. The left column illustrates the ground-truth heatmaps, showing where the vulnerability is present in the code, the center column brings the InputXGradient interpretability heatmaps, and the right column presents the Saliency interpretability heatmaps.

FIGURE 4.4 – Overview of the heatmaps for three vulnerable functions, being the left column the ground-truth, the center column the heatmap from the InputXGradient interpretability method, and the right column form the Saliency interpretability method. The importances given by the interpretability methods to each token are filtered using a threshold.

4.3.2 Comparison - CoTexT and CodeBERT

In this section, we assess two examples of interpretability heatmaps generated by the CodeBERT and CoTexT models, comparing them with the ground-truth, and discussing their differences and characteristics.

4.3.2.1 Example 1

Firstly, we examine a heatmap interpreting the prediction of the CodeBERT model, which is illustrated in Figure 4.5. The top figure shows the ground-truth mask, highlighting only the part of the code where the vulnerability is present. The code was tokenized utilizing the CodeBERT tokenizer, with the tokens separated by a space. The bottom figure shows the saliency heatmap, representing the importance the model gave to each token when making its final prediction. The importance values go from 0 to 1, with the higher values being represented by more opaque tones of red.

The first thing to notice is that the model gives some importance to every token in the code. Secondly, in many lines the token * is highlighted more strongly than the others. In C/C++ language, this symbol is used for pointer declaration, and many operations with pointers are present in vulnerabilities relating to the ill use of the computer memory. This observation suggests that the model learned this characteristic of the programming language by seeing pointers repeatedly present in vulnerabilities during its training.

Secondly, we observe that the token with the strongest importance in the heatmap, *alloc*, is in fact part of the vulnerability present in the code. This shows that the model paid attention to the right element to make its decision. Importantly, the highlighted token is part of the *malloc* function, which is used to allocate memory, and returns a pointer to the first byte of the allocated memory. This function, as in the case of pointer declaration, is also present in many known vulnerabilities relating to unsafe use of the computer memory, and shows another characteristic of the language the model has learned.

Figure 4.6 shows the same example for the CoTexT model. The top figure shows the ground-truth mask tokenized using the CoTexT tokenizer, and the bottom figure shows the importance the model gave to each token.

The first thing to observe is how different the code looks after passing through the CoTexT tokenizer in comparison to the same code passing through the CodeBERT tokenizer. Especially, we highlight that the CoTexT tokenizer does not present a token for curly braces, therefore the code does not contain any after tokenization. This detail is important once curly braces are a structural element of the programming language, and meaning may be lost in its absence. A second point that can be noted is that the CoTexT tokenizer breaks each word into more granular tokens, in comparison with the

FIGURE 4.5 – CodeBERT results. Top: Tokenized function using the CodeBERT tokenizer, with ground-truth mask highlighting the exact position of the vulnerability. Bottom: Heatmap of the importance/saliency given by the model to each token, in order for it to predict the code as vulnerable.

CodeBERT tokenizer. For instance, the word *stride* is broken into three tokens, *s-tri-de*, and the words *height* and *width* are broken into two tokens, *he-ight* and *wid-th*. For the CodeBERT tokenizer, in comparison, *stride* is broken into two tokens, while *height* and *width* remained a single token. In particular and most worrisome, the CoTexT tokenizer broke the function keyword *malloc* into two tokens: *mal-loc*. This finer granularity of separation can make it more difficult for the model to learn what is the meaning of particular elements of the language, making it harder for it to make an informed decision, least of all a decision that can be interpreted by a human. This conclusion meets our observation that the magnitude of the frequencies in the CoTexT histogram of importance scores was roughly twice that of CodeBERT, meaning more tokens for the same quantity of code.

These differences in the tokenizers of the deep learning models are hardly ever discussed, but in the case of interpreting the output of the models they seem to be of importance. In particular, the characteristics of the CoTexT tokenizer make the tokenized code harder to read by a human, do not make intuitive sense, and more importantly, may

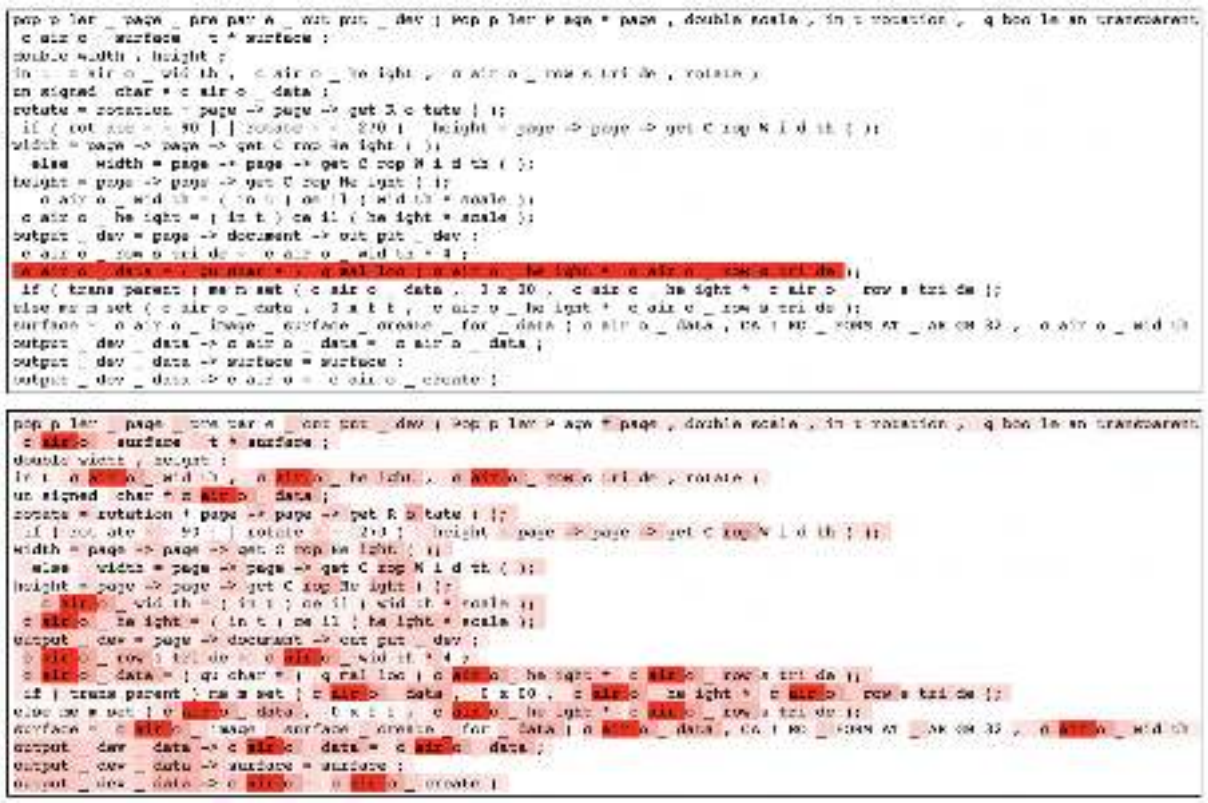


FIGURE 4.6 – CoTexT results. Top: Tokenized function using the CoTexT tokenizer, with ground-truth mask highlighting the exact position of the vulnerability. Bottom: Heatmap of the importance/saliency given by the model to each token, in order for it to predict the code as vulnerable.

cause the loss of crucial information.

Finally, it can be observed that the CoTexT model, as is the case for CodeBERT, gave some importance to all tokens in the code. But differently than for the CodeBERT model, the CoTexT model found one token to be very important: *air*, which forms the variable name *cairo*. This token was given high importance wherever it appeared, being spread over the code. We note that the variable name *cairo* appears in the line where the vulnerability is present, but once it is spread over the entirety of the code it is unclear if the model was looking at the right element when predicting the code as vulnerable. Additionally, once the interpretability method says the important tokens are spread in the code, it does not do much to help focusing the attention of the programmer on the right place where the vulnerability is.

4.3.2.2 Example 2

In the second example, we highlight other important details. The CodeBERT model result, illustrated in Figure 4.7 shows that, as in the previous example, the model gives some importance to all tokens. But again, the token with the most importance, **)*, is in the place where the vulnerability precisely is. It is also noticeable that the model does

not just go about giving high importance to any token representing a pointer allocation, once there are many such tokens in the code sample which are assigned small or moderate importance. This shows that the model has learned indeed how to discriminate the context of the tokens, and how the context is a better indicator of a possible vulnerability than just the token itself.

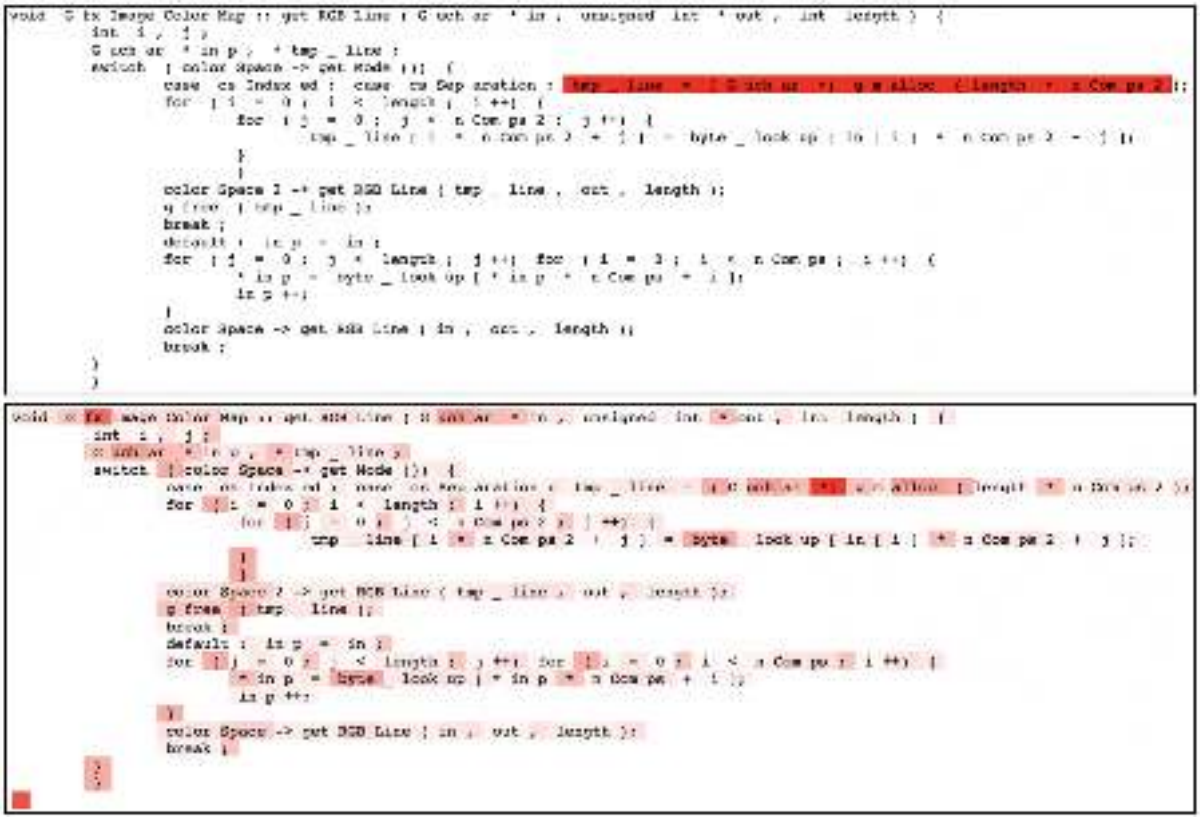


FIGURE 4.7 – CodeBERT results. Top: Tokenized function using the CodeBERT tokenizer, with ground-truth mask highlighting the exact position of the vulnerability. Bottom: Heatmap of the importance/saliency given by the model to each token, in order for it to predict the code as vulnerable.

The CoTexT result for the second example is illustrated in Figure 4.8. As in the first example, the CoTexT model has found some tokens to be important and highlighted them throughout the code, not giving any special attention to the region where the vulnerability really is. This shows that the model, in contrast with CodeBERT, lacks discriminative power, and again, the interpretability result does little to help the programmer to find where the vulnerability is located.

FIGURE 4.8 – CoTexT results. Top: Tokenized function using the CoTexT tokenizer, with ground-truth mask highlighting the exact position of the vulnerability. Bottom: Heatmap of the importance/saliency given by the model to each token, in order for it to predict the code as vulnerable.

5 Conclusion

In this work, we introduced a benchmark dataset of vulnerable functions with their corresponding ground-truth vulnerability masks, having the exact information of the position of the vulnerable code snippets. Additionally, we fine-tuned two state-of-the-art models for the task of vulnerability detection, CodeBERT and CoTexT, and utilized the InputXGradient and Saliency interpretability methods to extract the importance the models gave to each of their input tokens in order to arrive at their final prediction.

Firstly, we compared both interpretability methods, and we observed that they achieved similar mean precision results. This suggests that the Euclidean norm aggregation method employed by the InpuXGradient, even though ignoring the direction of the saliency vectors, does not seem to harm its performance. Additionally, we qualitatively compared the heatmaps produced by both methods, and concluded that the InputXGradient gives a considerably more interpretable heatmap, where it is easier to find the important pieces of code as seen by the model, as well as its correspondence with the ground-truth heatmap.

Secondly, comparing the interpretability heatmaps generated by both language models, we observed that even though the CoTexT model is the state-of-the-art for the vulnerability detection task, presenting slightly better accuracy in comparison with CodeBERT, its interpretability heatmap is farther from the truth than CodeBERT's by a considerable margin.

The CodeBERT interpretability heatmap brings more concise information, highlighting fewer tokens as the most important, which may help programmers to focus their vulnerability search, besides being more precise. CodeBERT also showed good signs of having learned important language characteristics, like the roles of pointer declaration and memory allocation. Finally, CodeBERT was able to show discriminative power, giving different importances scores for different appearances of the same token, based on its context.

Contrastively, the CoTexT model presented a tendency of giving high importance to the same token throughout the code, which showed lack of discrimination and is a feature of little help for the programmer trying to fix a vulnerable code. Additionally, we observed characteristics of the CoTexT tokenizer that can make it harder for the model to learn

syntactic and semantic meaning of the language, or even impossible for it to make the right prediction: the model vocabulary did not contained structural tokens of the language (e.g., curly braces), and the tokenizer broke the code strings into very granular pieces, separating important keywords in meaningless tokens (e.g., *malloc* into *mal-loc*).

This was the first work in the field of vulnerability detection, to the authors' knowledge, to take a more systematic look at the state-of-the-art models together with interpretability techniques. It was noticeable during this study the importance the tokenizer has in order to make it possible for the language model to pay attention to the right elements when making a prediction, and also to make its prediction favorable to human interpretation. Additionally, we note the stark variation in the output of the different interpretability methods tested, these having a variable degree of quality and clearness.

The dataset introduced here can be further used in order to evaluate the interpretability capacity of other deep learning models, as well as evaluate and improve the interpretability methods available.

This is an important step for the deployment in practice of deep learning models trained in the task of vulnerability detection, which could be a great improvement in productivity if these models can show precisely where the vulnerabilities are present in the code. Above all, the application of such models, due to their power, can represent a great gain in code safety.

5.1 Contributions

This study has made the following contributions to the field:

- A novel benchmark dataset of vulnerable functions with their corresponding ground-truth vulnerability masks;
- A systematic look at interpretability techniques for vulnerability detection, and their behavior when coupled with state of the art language models.

5.2 Future works

The work presented here is but a first effort for the development of a more systematic approach for the application of interpretability methods for vulnerability detection in programming languages. The field is currently wide open, and we list topics that could be explored in future works:

1. A limitation of using precision as the evaluation metric is that it does not take into account the length of the vulnerability in the input function (in terms of tokens). The consequence is that the result may be biased toward functions with smaller vulnerability length, in which it may be easier to attain a high precision score. This drawback can be addressed by the use of a precision metric normalized by the length of the vulnerability in the code.

2. One possible avenue of exploration in order to try and improve the current interpretability results is to implement methods of pre-processing the function before feeding it to the language models, and/or post-processing the importance heatmaps obtained from the interpretability methods. One possible pre-processing approach could be, for instance, to replace the variable names for placeholders. The post-processing, on the other hand, could be implemented by removing the importance of tokens which should not be of any real importance for security vulnerabilities, such as the comma or hyphen symbols, and can be reducing the precision of the method.

3. We limited the scope of our work to only two interpretability methods, which were shown to present the best results in Natural Language Processing tasks (ATANASOVA *et al.*, 2020). However, there is a growing plethora of interpretability methods available, and an exploration and comparison of these different methods in programming language tasks could give us a better understanding of what are the best-performing and more adequate methods for application in this field.

4. It was discussed the limitation the interpretability method InputXGradient presents when coupled with Euclidean norm aggregation, due to the fact that it gives the same importance to two tokens that have the same saliency vector magnitude, even if their vectors are pointing to different directions. The development of aggregation techniques that take advantage of the direction of the saliency vector as well as its magnitude may be an important step toward better interpretability methods.

5. Benchmark datasets for the task of vulnerability detection which are large enough to train or finetune deep learning models are currently limited to C/C++ languages. This represents a major bottleneck to the practical application of the models trained. One important step, though tedious and time-consuming, is the curation of more diverse datasets, encompassing more programming languages.

6. Adding to the previous item, one limitation of the current datasets used for vulnerability detection is that they have only information whether a function is vulnerable or not, but not what is the nature or kind of the vulnerability. A dataset with a diverse set of vulnerability categories would be a rich resource in order to train models with useful outputs for real-life applications.

Bibliography

AHMAD, W. U.; CHAKRABORTY, S.; RAY, B.; CHANG, K.-W. **Unified Pre-training for Program Understanding and Generation**. arXiv, 2021. Available at: https://arxiv.org/abs/2103.06333.

ATANASOVA, P.; SIMONSEN, J. G.; LIOMA, C.; AUGENSTEIN, I. **A Diagnostic Study of Explainability Techniques for Text Classification**. arXiv, 2020. Available at: https://arxiv.org/abs/2009.13295.

BURATTI, L.; PUJAR, S.; BORNEA, M.; MCCARLEY, S.; ZHENG, Y.; ROSSIELLO, G.; MORARI, A.; LAREDO, J.; THOST, V.; ZHUANG, Y.; DOMENICONI, G. **Exploring Software Naturalness through Neural Language Models**. arXiv, 2020. Available at: https://arxiv.org/abs/2006.12641.

BURKART, N.; HUBER, M. F. **A Survey on the Explainability of Supervised Machine Learning**. AI Access Foundation, jan 2021. 245–317 p. Available at: https://doi.org/10.16132Fjair.1.12228.

CASTRO, J.; GóMEZ, D.; TEJADA, J. **Polynomial calculation of the Shapley value based on sampling**. 2009. 1726-1730 p. Selected papers presented at the Tenth International Symposium on Locational Decisions (ISOLDE X). Available at: https://www.sciencedirect.com/science/article/pii/S0305054808000804.

CHEIRDARI, F.; KARABATIS, G. **Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools**. 2018. 4782-4788 p.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. d. O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G.; RAY, A.; PURI, R.; KRUEGER, G.; PETROV, M.; KHLAAF, H.; SASTRY, G.; MISHKIN, P.; CHAN, B.; GRAY, S.; RYDER, N.; PAVLOV, M.; POWER, A.; KAISER, L.; BAVARIAN, M.; WINTER, C.; TILLET, P.; SUCH, F. P.; CUMMINGS, D.; PLAPPERT, M.; CHANTZIS, F.; BARNES, E.; HERBERT-VOSS, A.; GUSS, W. H.; NICHOL, A.; PAINO, A.; TEZAK, N.; TANG, J.; BABUSCHKIN, I.; BALAJI, S.; JAIN, S.; SAUNDERS, W.; HESSE, C.; CARR, A. N.; LEIKE, J.; ACHIAM, J.; MISRA, V.; MORIKAWA, E.; RADFORD, A.; KNIGHT, M.; BRUNDAGE, M.; MURATI, M.; MAYER, K.; WELINDER, P.; MCGREW, B.; AMODEI, D.; MCCANDLISH, S.; SUTSKEVER, I.; ZAREMBA, W. **Evaluating Large Language Models Trained on Code**. arXiv, 2021. Available at: https://arxiv.org/abs/2107.03374.

DAM, H. K.; TRAN, T.; PHAM, T.; NG, S. W.; GRUNDY, J.; GHOSE, A. **Automatic Feature Learning for Predicting Vulnerable Software Components**. 2021. 67-85 p.

DEVLIN, J.; CHANG, M.-W.; LEE, K.; TOUTANOVA, K. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. arXiv, 2018. Available at: https://arxiv.org/abs/1810.04805.

ERHAN, D.; BENGIO, Y.; COURVILLE, A.; MANZAGOL, P.-A.; VINCENT, P.; BENGIO, S. **Why Does Unsupervised Pre-training Help Deep Learning?** 2010. 625–660 p. Available at: http://jmlr.org/papers/v11/erhan10a.htmlhttp://jmlr.org/papers/v11/erhan10a.html.

FACELI, K.; LORENA, A. C.; GAMA, J.; CARVALHO, A. C. P. de Leon Ferreira de. **Inteligência Artificial : Uma Abordagem de Aprendizagem de Máquina**. [*S.l.*]: LTC, 2011.

FAN, J.; LI, Y.; WANG, S.; NGUYEN, T. N. **A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries**. New York, NY, USA: Association for Computing Machinery, 2020. 508–512 p. (MSR '20). Available at: https://doi.org/10.1145/3379597.3387501.

FENG, Z.; GUO, D.; TANG, D.; DUAN, N.; FENG, X.; GONG, M.; SHOU, L.; QIN, B.; LIU, T.; JIANG, D.; ZHOU, M. **CodeBERT: A Pre-Trained Model for Programming and Natural Languages**. arXiv, 2020. Available at: https://arxiv.org/abs/2002.08155.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [*S.l.*]: MIT Press, 2016. `http://www.deeplearningbook.org`.

HANIF, H.; MAFFEIS, S. **VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection**. arXiv, 2022. Available at: https://arxiv.org/abs/2205.12424.

HINDLE, A.; BARR, E. T.; SU, Z.; GABEL, M.; DEVANBU, P. **On the Naturalness of Software**. [*S.l.*]: IEEE Press, 2012. 837–847 p. (ICSE '12).

INC., S. S. **Rough Audit Tool for Security. https://github.com/stgnet/rats**. 2013.

KIM, Y. **Convolutional Neural Networks for Sentence Classification**. arXiv, 2014. Available at: https://arxiv.org/abs/1408.5882.

KINDERMANS, P.-J.; HOOKER, S.; ADEBAYO, J.; ALBER, M.; SCHüTT, K. T.; DäHNE, S.; ERHAN, D.; KIM, B. **The (Un)reliability of saliency methods**. arXiv, 2017. Available at: https://arxiv.org/abs/1711.00867.

KINDERMANS, P.-J.; SCHüTT, K.; MüLLER, K.-R.; DäHNE, S. **Investigating the influence of noise and distractors on the interpretation of neural networks**. arXiv, 2016. Available at: https://arxiv.org/abs/1611.07270.

KING, J. C. **Symbolic Execution and Program Testing**. New York, NY, USA: Association for Computing Machinery, jul 1976. 385–394 p. Available at: https://doi.org/10.1145/360248.360252.

LIU, T.; CHEN, M.; ZHOU, M.; DU, S. S.; ZHOU, E.; ZHAO, T. **Towards Understanding the Importance of Shortcut Connections in Residual Networks**. arXiv, 2019. Available at: https://arxiv.org/abs/1909.04653.

LIU, Y.; OTT, M.; GOYAL, N.; DU, J.; JOSHI, M.; CHEN, D.; LEVY, O.; LEWIS, M.; ZETTLEMOYER, L.; STOYANOV, V. **RoBERTa: A Robustly Optimized BERT Pretraining Approach**. arXiv, 2019. Available at: https://arxiv.org/abs/1907.11692.

LU, S.; GUO, D.; REN, S.; HUANG, J.; SVYATKOVSKIY, A.; BLANCO, A.; CLEMENT, C.; DRAIN, D.; JIANG, D.; TANG, D.; LI, G.; ZHOU, L.; SHOU, L.; ZHOU, L.; TUFANO, M.; GONG, M.; ZHOU, M.; DUAN, N.; SUNDARESAN, N.; DENG, S. K.; FU, S.; LIU, S. **CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation**. arXiv, 2021. Available at: https://arxiv.org/abs/2102.04664.

MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. **Efficient Estimation of Word Representations in Vector Space**. arXiv, 2013. Available at: https://arxiv.org/abs/1301.3781.

MOSOLYGó, B.; VáNDOR, N.; HEGEDUNDEFINEDS, P.; FERENC, R. **A Line-Level Explainable Vulnerability Detection Approach for Java**. Berlin, Heidelberg: Springer-Verlag, 2022. 106–122 p. Available at: https://doi.org/10.1007/978-3-031-10542-5_8.

NGUYEN, V.-A.; NGUYEN, D. Q.; NGUYEN, V.; LE, T.; TRAN, Q. H.; PHUNG, D. **ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection**. arXiv, 2021. Available at: https://arxiv.org/abs/2110.07317.

PENNINGTON, J.; SOCHER, R.; MANNING, C. **GloVe: Global Vectors for Word Representation**. Doha, Qatar: Association for Computational Linguistics, out. 2014. 1532–1543 p. Available at: https://aclanthology.org/D14-1162.

PHAN, L.; TRAN, H.; LE, D.; NGUYEN, H.; ANIBAL, J.; PELTEKIAN, A.; YE, Y. **CoTexT: Multi-task Learning with Code-Text Transformer**. arXiv, 2021. Available at: https://arxiv.org/abs/2105.08645.

RADFORD, A.; NARASIMHAN, K. **Improving Language Understanding by Generative Pre-Training**. 2018.

RAFFEL, C.; SHAZEER, N.; ROBERTS, A.; LEE, K.; NARANG, S.; MATENA, M.; ZHOU, Y.; LI, W.; LIU, P. J. **Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer**. arXiv, 2019. Available at: https://arxiv.org/abs/1910.10683.

RENIERES, M.; REISS, S. **Fault localization with nearest neighbor queries**. 2003. 30-39 p.

RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. **"Why Should I Trust You?": Explaining the Predictions of Any Classifier**. arXiv, 2016. Available at: https://arxiv.org/abs/1602.04938.

RUSSAKOVSKY, O.; DENG, J.; SU, H.; KRAUSE, J.; SATHEESH, S.; MA, S.; HUANG, Z.; KARPATHY, A.; KHOSLA, A.; BERNSTEIN, M.; BERG, A. C.; FEI-FEI, L. **ImageNet Large Scale Visual Recognition Challenge**. arXiv, 2014. Available at: https://arxiv.org/abs/1409.0575.

RUSSELL, R. L.; KIM, L.; HAMILTON, L. H.; LAZOVICH, T.; HARER, J. A.; OZDEMIR, O.; ELLINGWOOD, P. M.; MCCONLEY, M. W. **Automated Vulnerability Detection in Source Code Using Deep Representation Learning**. arXiv, 2018. Available at: https://arxiv.org/abs/1807.04320.

SILVEIRA, L.; MARCONDES, C. A. C.; VERRI, F. A. N. **Programming Language Vulnerability Detection and Interpretability with Language Models**. jun. 2023. Available at: https://doi.org/10.5281/zenodo.7863261.

SIMONYAN, K.; VEDALDI, A.; ZISSERMAN, A. **Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps**. arXiv, 2013. Available at: https://arxiv.org/abs/1312.6034.

SOTGIU, A.; PINTOR, M.; BIGGIO, B. **Explainability-Based Debugging of Machine Learning for Vulnerability Discovery**. New York, NY, USA: Association for Computing Machinery, 2022. (ARES '22). Available at: https://doi.org/10.1145/3538969.3543809.

SPRINGENBERG, J. T.; DOSOVITSKIY, A.; BROX, T.; RIEDMILLER, M. **Striving for Simplicity: The All Convolutional Net**. arXiv, 2014. Available at: https://arxiv.org/abs/1412.6806.

SUNDARARAJAN, M.; TALY, A.; YAN, Q. **Axiomatic Attribution for Deep Networks**. arXiv, 2017. Available at: https://arxiv.org/abs/1703.01365.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L.; POLOSUKHIN, I. **Attention Is All You Need**. arXiv, 2017. Available at: https://arxiv.org/abs/1706.03762.

WAN, Y.; ZHAO, W.; ZHANG, H.; SUI, Y.; XU, G.; JIN, H. **What Do They Capture? A Structural Analysis of Pre-Trained Language Models for Source Code**. New York, NY, USA: Association for Computing Machinery, 2022. 2377–2388 p. (ICSE '22). Available at: https://doi.org/10.1145/3510003.3510050.

WATTANAKRIENGKRAI, S.; THONGTANUNAM, P.; TANTITHAMTHAVORN, C.; HATA, H.; MATSUMOTO, K. **Predicting Defective Lines Using a Model-Agnostic Technique**. 2022. 1480-1496 p.

WHEELER, D. A. **Flawfinder. http://www.dwheeler.com/flawfinder**. 2018.

XU, J.; SUN, X.; ZHANG, Z.; ZHAO, G.; LIN, J. **Understanding and Improving Layer Normalization**. arXiv, 2019. Available at: https://arxiv.org/abs/1911.07013.

YAMAGUCHI, F.; GOLDE, N.; ARP, D.; RIECK, K. **Modeling and Discovering Vulnerabilities with Code Property Graphs**. 2014. 590-604 p.

ZEILER, M. D.; FERGUS, R. **Visualizing and Understanding Convolutional Networks**. arXiv, 2013. Available at: https://arxiv.org/abs/1311.2901.

ZHOU, Y.; LIU, S.; SIOW, J.; DU, X.; LIU, Y. **Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks**. arXiv, 2019. Available at: https://arxiv.org/abs/1909.03496.

Annex A - Publications

During the master's program, the following papers have been published or submitted by the author:

•Published:

1. Leonardo Silveira, Matheus Rodrigues, Bruno Faiçal, Alexandre Silva, Cesar Marcondes, Marcos R. O. A. Maximo, Filipe Verri, "Navigation Aids based on Optical Flow and Convolutional Neural Network", LARS-SBR 2022 Proceedings, 18-21 October 2022. https://doi.org/10.1109/LARS/SBR/WRE56824.2022.9995889
2. Ronaldo Júnior, Leonardo Silveira, Victor Castro Nacif de Faria, Ana Carolina Lorena, "Justiça nas previsões de modelos de Aprendizado de Máquina: um estudo de caso com dados de reincidência criminal", ENIAC 2022 Proceedings, 28-1 November 2022. https://doi.org/10.5753/eniac.2022.227610

•Submitted:

1. Paulo Victor Lopes, Leonardo Silveira, Roberto Douglas Guimaraes Aquino, Carlos Henrique Ribeiro, Anders Skoogh, Filipe Alves Neto Verri, "Synthetic Discrete Event Data Generation For Digital Twins: Enabling Production Systems Analysis in the Absence of Data", Submitted to International Journal of Computer Integrated Manufacturing, 2023.

Additionally, the paper derived from this Master's thesis is being prepared for submission to the Journal of Machine Learning Research.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. REGISTRO N°	4. N° DE PÁGINAS
DM	26 de junho de 2023	DCTA/ITA/DM-030/2023	61

5. TÍTULO E SUBTÍTULO:

Source code vulnerability detection and interpretability with language models.

6. AUTOR(ES):

Leonardo Silveira

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Vulnerability Detection; Interpretability; Pre-Trained Language Models; Deep Learning

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Processamento de linguagem natural; Vulnerabilidade; Códigos-fonte; Análise de tendências; Aprendizagem (inteligência artificial); Banco de dados; Computação.

10. APRESENTAÇÃO: **(X) Nacional** **( ) Internacional**

ITA, São José dos Campos. Curso de Mestrado. Programa de Pós-Graduação em Engenharia Eletrônica e Computação. Área de Informática. Orientador: Filipe Alves Neto Verri; co-orientador: Cesar Augusto Cavalheiro Marcondes. Defesa em 31/05/2023. Publicada em 2023.

11. RESUMO:

The execution of security tests in software to detect vulnerabilities is fundamental in order to avoid the occurrence of malicious attacks, which can, among other things, compromise the operation of the application or expose sensitive data of its users. Traditionally, these tests are performed using static, dynamic or symbolic analysis tools. These tools have several limitations, such as the detection of many false positives and very high computational cost. An alternative to traditional methods is the use of machine learning, particularly deep learning models. Several approaches exploring these models have been proposed in recent literature, mostly inspired by the advances achieved in deep machine learning in the field of Natural Language Processing. Despite the use of increasingly sophisticated models, little has been done in the field of interpretability of these methods, with the aim of reaching not only the classification of the source code as vulnerable or not, but also to point out in the code the passages in which the learning model believes that the vulnerability is present. In this work, we perform the training (fine-tuning) of two language models, CodeBERT and CoTexT, for the task of detecting vulnerabilities in source codes, and we evaluate the ability of these models to generate interpretation of their predictions. For this, we curated a database composed of a collection of vulnerable codes and their respective labeling masks, locating the exact position of the vulnerabilities in the code. Using the two best interpretability methods for text classification tasks in Natural Language Processing, Salience and InputXGradient, we generated heat maps representing the importance of each code token for model prediction. Thus, we found that both techniques present similar precision results, however the heat maps generated by the InputXGradient method are considerably easier to be interpreted. Comparing the language models, CodeBERT presents slightly lower accuracy when compared to CoTexT in the source code classification task, however, in contrast, it generates considerably better results of interpretability of its prediction. Additionally, we highlight the effect of tokenizers used by language models on their ability to generate interpretation of their predictions, showing that their characteristics can significantly influence the model's ability to learn the syntactic and semantic structure of language.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) SECRETO**