SCUOLA DI DOTTORATO
Università degli Studi di Milano-Bicocca

Department of Informatics, Systems and Communication

PhD program in Computer Science — Cycle XXXV

ARTIFICIAL INTELLIGENCE FOR PROGRAM COMPREHENSION: DISCLOSING NEURAL MODELS TRAINED ON SOURCE CODE

MARTINA SALETTA
736106

Supervisor: Prof. Claudio Ferretti

Tutor: Prof. Leonardo Mariani
Coordinator: Prof. Leonardo Mariani

Academic Year 2021-2022

Martina Saletta: *Artificial Intelligence for Program Comprehension: Disclosing Neural Models Trained on Source Code,* A.Y. 2021-2022

ABSTRACT

The field of Artificial Intelligence (AI) has made significant advancements in recent years and has been applied to various domains including that of program comprehension. In this thesis, the main interest consists in devising AI approaches to deal with source code, in order to be able to extract knowledge from it. The featured work is developed along two main research lines, namely 1. the automatic recognition of particular source code properties which can range from the presence of particular syntactic patterns or constructs to higher level features such as the functional purpose of code snippets, and 2. the study of how existing neural models trained in different source code-related tasks make their predictions. In this case, the main goal is to investigate which elements are mostly involved in the decision process of a network.

The carried out work allowed to obtain insightful research outcomes, including the design of a source code embedding able to capture specific properties, a thorough analysis of the internal dynamics of neural networks trained on source code, the investigation of the decision process of state of the art neural transformers in terms of identification of which source code features, or high-level concepts, are more relevant to make a prediction, and the definition of a method to synthesise adversarial instances able to deceive a network.

RIASSUNTO

L'intelligenza artificiale ha avuto un grande sviluppo negli ultimi anni, ed è stata applicata a vari domini, tra cui quello della program comprehension. L'obiettivo principale di questa tesi è quello di sviluppare approcci basati sull'intelligenza artificiale che operino su codice sorgente, in modo da poterne estrarre conoscenza. Il lavoro si sviluppa su due principali linee, ovvero 1. il riconoscimento automatico di particolari proprietà che possono spaziare dalla presenza di specifici pattern sintattici a caratteristiche più di alto livello, come l'obiettivo funzionale di frammenti di codice, e 2. lo studio di come modelli neurali esistenti addestrati per differenti scopi fanno le loro predizioni. In questo caso, l'obiettivo principale è quello di analizzare quali elementi sono più coinvolti nel processo di decisione della rete. Il lavoro sviluppato ha permesso di ottenere risultati promettenti, tra cui lo sviluppo di un embedding per codice sorgente capace di cogliere diverse proprietà, una profonda analisi delle dinamiche interne di reti neurali addestrate su codice sorgente, lo studio del processo di decisione di moderni transformer in termini di identificazione di caratteristiche, o concetti ad alto livello, sono più importanti per la predizione, e la definizione di un metodo per sintetizzare istanze adversarial per ingannare una rete neurale.

PUBLICATIONS

Some of the results and ideas contained in this thesis have been previously appeared in the following works that are currently published (•) or submitted (∗):

- M. Saletta, C. Ferretti. *A Neural Embedding for Source Code: Security Analysis and CWE Lists*. In: Proceedings of 18th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC) DASC-PICom-CBDCom-CyberSciTech, pp. 529-536. 2020.
- C. Ferretti, M. Saletta. *Deceiving Neural Source Code Classifiers: Finding Adversarial Examples With Grammatical Evolution*. In: Proceedings of GECCO 21 The Genetic and Evolutionary Computation Conference, Companion Volume, pp. 1889-1897. 2021.
- M. Saletta, C. Ferretti. *Towards the Evolutionary Assessment of Neural Transformers Trained on Source Code*. In: Proceedings of GECCO 22 The Genetic and Evolutionary Computation Conference, Companion Volume, pp. 1770-1778. 2022.
- M. Saletta, C. Ferretti. *A Grammar-based Evolutionary Approach for Assessing Deep Neural Source Code Classifiers*. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC). 2022.
- [1]C. Ferretti, M. Saletta. *Do Neural Transformers Learn Human Defined Concept? An Extensive Study Source Code Processing Domain.* In: Algorithms 15(12):449. 2022.
- ∗ M. Saletta, C. Ferretti. *Exploring Neural Dynamics in Source Code Processing Domain*. Submitted to Information.

Below a list of other works published (•) or submitted (∗) during the doctorate, but whose content is not discussed in the thesis:

- L. Mariot, M. Saletta, A. Leporati, L. Manzoni. *Exploring Semi-bent Boolean Functions Arising from Cellular Automata*. In: Proceedings of 14th International Conference on Cellular Automata for Research and Industry (ACRI), pp. 56-66. 2020.
- L. Mariot, M. Saletta, A. Leporati, L. Manzoni. *Heuristic Search of (Semi-)Bent Functions based on Cellular Automata*. In: Natural Computing 21(3), pp. 377-391. 2022.
- ∗ C. Ferretti, M. Saletta. *Naturalness in Source Code Summarization. How Significant is it?* Submitted to 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023 [accepted].

1 Selected as cover story of *Algorithms*, Volume 15, issue 12 (December 2022)

CONTENTS

I PRELIMINARIES 1

1 INTRODUCTION 3
1.1 Thesis Outline . . . 4

2 LITERATURE OVERVIEW 7
2.1 Machine Learning for Source Code Analysis . . . 7
2.1.1 Source Code Embedding . . . 8
2.1.2 Vulnerability Detection . . . 9
2.2 Formal Grammars and Evolutionary Algorithms . . . 10
2.2.1 Evolutionary Program Synthesis . . . 10
2.3 Explaining Machine Learning Models . . . 12
2.3.1 Classification Errors . . . 12
2.3.2 Analysis of Single Neurons . . . 12
2.3.3 Saliency, Features and Concepts . . . 13
2.3.4 Adversarial Approaches . . . 13

3 BACKGROUND NOTIONS 15
3.1 Transformers for Source Code Processing . . . 15
3.2 Grammar-based Evolutionary Algorithms . . . 17
3.2.1 Grammatical Evolution . . . 18
3.2.2 Dynamic Structured Grammatical Evolution . . . 20

II RESEARCH 23

4 A NEURAL SOURCE CODE EMBEDDING 25
4.1 Introduction and Motivations . . . 25
4.2 Embedding Design . . . 26
4.3 Experimental Settings . . . 27
4.3.1 Preprocessing . . . 28
4.3.2 Training . . . 29
4.3.3 Inference . . . 29
4.4 Embedding evaluation . . . 31
4.4.1 Cluster Analysis . . . 32
4.4.2 Qualitative Assessment . . . 34
4.4.3 Supervised Classification . . . 37
4.5 Discussion . . . 40
4.5.1 CWE Views . . . 40
4.5.2 Significance and Limitations . . . 40
4.5.3 Further Directions . . . 41

5 NEURAL DYNAMICS AND CONCEPT MINING 43
5.1 Introduction . . . 43
5.2 Approach Overview . . . 44
5.3 Experimental Settings . . . 46
5.3.1 Network Architecture . . . 46
5.3.2 Dataset and Training . . . 46
5.4 Task-based Experiments . . . 47
5.4.1 Problems Definition . . . 48
5.4.2 Classification . . . 50

5.5 Neurons Ranking . . . 52
5.5.1 Entropy of Single Neurons . . . 52
5.5.2 Pairwise comparison . . . 53
5.6 Results . . . 54
5.6.1 Classification Experiments . . . 54
5.6.2 Entropy-based Experiments . . . 55
5.6.3 Pairwise Comparison Experiments . . . 57
5.7 Final Remarks . . . 59
5.7.1 Insights and Limitations . . . 60
5.7.2 Further Directions . . . 60
6 EVOLUTIONARY APPROACHES FOR ADVERSARIAL ATTACKS 63
6.1 Robustness and Adversarial Attacks . . . 63
6.2 Work Overview . . . 65
6.3 Experiments . . . 66
6.3.1 Vulnerability Detection Model . . . 66
6.3.2 Grammatical Evolution . . . 69
6.3.3 Evolutionary Runs . . . 70
6.3.4 Classification of Evolved Instances . . . 71
6.4 Discussion . . . 72
6.4.1 Threat Model . . . 72
6.4.2 Neural Fitness Functions: Further Perspectives . 74
7 INPUT SPACE SAMPLING AND DECISION BOUNDARIES 75
7.1 Introduction . . . 75
7.2 Methods . . . 76
7.2.1 The CuBERT Source Code Classifier . . . 77
7.3 Two-stage Evolutionary Search . . . 77
7.3.1 DSGE and Neural Fitness Functions . . . 78
7.3.2 Mutations and Fitness Variations . . . 79
7.4 Results . . . 81
7.4.1 Sampling the solution space . . . 81
7.4.2 Moving across decision boundaries . . . 82
7.4.3 Blind Spots and Salient Features . . . 85
7.5 Discussion . . . 87
8 CONCEPT-BASED EXPLAINABILITY 89
8.1 Motivations and Work Overview . . . 89
8.2 Approach Description . . . 90
8.2.1 Input instances and sub-concepts . . . 90
8.2.2 Activations space, linear SVCs and concept-based neural fitness function . . . 91
8.2.3 Sensitivity to sub-concepts . . . 93
8.3 Experiments . . . 93
8.3.1 Data Preparation and Fine-tuning . . . 94
8.3.2 Sub-concepts formulation . . . 95
8.3.3 SVCs and activations spaces . . . 96
8.3.4 Evolutionary search along sub-concepts directions 97
8.3.5 Measuring sensitivity to sub-concepts . . . 98
8.4 Results . . . 99
8.5 Discussion and Final Remarks . . . 101

III CLOSING REMARKS 103

9 CONCLUSIONS 105
9.1 Outcomes Summary 105
9.1.1 AST-based source code embedding 105
9.1.2 Analysis of single internal neurons 106
9.1.3 Evolving adversarial input instances 106
9.1.4 Evolving along *concept* directions 106
9.1.5 Salient features 107
9.2 Further Research Directions 107
9.2.1 "Augmented" programming 107
9.2.2 Source code summarization 108

BIBLIOGRAPHY 111

LIST OF FIGURES

Figure 1 Transformer architecture 16
Figure 2 Example of genotype to phenotype mapping in GE 20
Figure 3 DSGE overview 21
Figure 4 DSGE genotype to phenotype decoding 22
Figure 5 Embedding generation outline 26
Figure 6 Node to word conversion example 28
Figure 7 Silhouette of different models 31
Figure 8 Silhouette for weaknesses categories 33
Figure 9 2D visualization of vectors representing different weaknesses. 35
Figure 10 CWE classification accuracy of different supervised algorithms 37
Figure 11 SVM confusion matrix 39
Figure 12 Autoencoder architecture 46
Figure 13 Neurons' classification accuracy 56
Figure 14 Distribution of the neurons' entropies 56
Figure 15 Activations of neurons having different entropies 56
Figure 16 Comparison between high entropy and low entropy neurons 57
Figure 17 Best neuron in a classification task compared with neurons performing bad on the same task 57
Figure 18 Ratio of successful Mann-Whitney U tests . . . 58
Figure 19 Attack scenario 64
Figure 20 Evolution of pure individuals 67
Figure 21 Evolution of hybrid individuals 67
Figure 22 Classification experiments 67
Figure 23 Example of individual evolved with GE (minimized) 68
Figure 24 Example of individual evolved with GE (maximized) 68
Figure 25 BNF C grammar to evolve adversarial instances 68
Figure 26 Fitness of the evolved individuals 71
Figure 27 Simplified Python grammar 78
Figure 28 Fitness values over the DSGE runs 83
Figure 29 Results of the mutation based experimental phase 83
Figure 30 Examples for the mutation based experimental phase 84
Figure 31 Concept-based experiments overview 91
Figure 32 Simplified Java grammar 94
Figure 33 CuBERT encoder block 96
Figure 34 Fitness of instances evolved with DSGE 98
Figure 35 SVC results 99
Figure 36 SVC results (random) 100

LIST OF TABLES

Table 1 Properties of the trained ast2vec models 30
Table 2 Summary of the CWE categories 30
Table 3 Silhouette score obtained by different models . 32
Table 4 Statistics of the silhouette score obtained for different CWEs 34
Table 5 Description of the discussed CWE 36
Table 6 Single CWEs considered for classification . . . 38
Table 7 Best accuracy score for each problem instance 51
Table 8 Classification results 73
Table 9 Confusion matrix (Original) 73
Table 10 Confusion matrices (Modified) 73
Table 11 Sensitivity to different concepts 100
Table 12 Sensitivity of the models to different strength of concepts 102

ACRONYMS

AST	Abstract Syntax Tree
BNF	Backus-Naur Form or Backus Normal Form
CAV	Concept Activation Vector
CFG	Context Free Grammar
CWE	Common Weakness Enumeration
DSGE	Dynamic Structured Grammatical Evolution
EA	Evolutionary Algorithm
GE	Grammatical Evolution
GP	Genetic Programming
ML	Machine Learning
NLP	Natural Language Processing
SGE	Structured Grammatical Evolution
SVC	Support Vector Classifier
TCAV	Testing with Concept Activation Vectors

Part I

PRELIMINARIES

1 INTRODUCTION

Program comprehension, in its historical and standard connotation [21, 96], concerns the studies performed in the field of software engineering for providing effective software maintenance and evolution. In general, it consists in studying how programmers comprehend the source code they are working on, by analysing, for instance, the cognitive processes behind tasks such as reading and understanding source code written by other programmers, and in the design of representations or approaches that can ease the programmers in activities such as writing, correcting or reusing source code. Classical examples [24] are aimed at defining cognitive complexity models, and at easing the effort required to understand source code when performing related processes such as those of *chunking* and *tracing*. In general, given the underneath goal to help the work of software developers, the studies in the field of program comprehension often needed a validation on groups of programmers, in some cases also with the aid of sophisticated facilities such as functional magnetic resonance imaging (fMRI) [133].

In recent years, the diffusion of machine learning approaches allowed to automatize the analysis of artifacts such as images or texts. These kind of analysis concerned a wide variety of objectives, from sentiment analysis [99] in natural language processing (NLP) to image understanding and pattern recognition [64]. One of the key aspects that is common to all these approaches, is that they are conceived to automatically extract some information from the object being analysed. With the growing impact that artificial intelligence (AI) systems – and especially those based on artificial neural networks – are having in everyday lives, also the way how these automation processes in the field of source code analysis are conceptualised and designed is inevitably changing, and thus their applicability, usefulness and purposes are undergoing a radical transformation. Recent works in the field of program comprehension are related to more *semantic* tasks instead of strictly engineering ones, such as source code summarization [153], feature location [41], or code generation [136]. Therefore, in general, besides the traditional goal of assisting the developers in their programming works, also the development of AI models that can somehow substitute (or significantly assist) the humans is becoming very common.

The ideas that underlie the work developed in this thesis are set in this context. Starting from the program comprehension notion as set out above, the main interest is to devise new ways to deal with source code, in order to be able to extract knowledge from it. Following the trend that leads the recent scientific research in many areas, the investigations will be based on artificial neural networks, and the goals will be manifold:

- the design of source code representations that are able to capture specific properties, so as to possibly use them to feed neural models to ease their work and to strengthen their robustness and reliability;
- the study of approaches to investigate the internal dynamics of the neural networks. The aim is to enhance the knowledge that can be obtained from a network, by adding information to that which naturally derives from the inference done with respect to the original training task;
- the investigation of the decision process of state of the art networks in terms of identification of which source code features, or high-level concepts, are more relevant for a network to make a prediction. The effect is to obtain evidence on possible misconceptions or blind-spots raised in the training phase;
- the exploitation of these evidence to devise methods to synthesise instances able to deceive a network, with the aim to attack the models from an adversarial perspective.

1.1 THESIS OUTLINE

This section provides a road-map for the reading of this thesis, which presents a progress in the field of program comprehension, intended in its meaning and purposes as described above. First, the literature overview of Chapter 2 introduces the topics involved in the discussion in terms of approaches and key results. Due to the range of diverse subjects covered, it is not to be intended as a systematic review, but more as a summary that is useful to guide the reader in the broad scientific scenario of the topics involved, and to focus on the perspective that concerns the study. Then, Chapter 3 moves to a more technical standpoint, and describes some of the models and algorithms used to design the experiments.

With Chapter 4 begins the core part of the thesis, which contains the studies and the research work carried out during the PhD, to investigate and explore the ideas outlined in the introduction. Specifically, Chapter 4 illustrates the conceptualization and design of a general purpose source code embedding (namely, a vector representation for source code able to capture and to preserve in the embedding space some semantic properties), and details the experimentation performed as a proof of concept, which points out its effectiveness when used as input for supervised classification models. Chapter 5 presents an extensive study on how the internal dynamics of neural networks trained on source code can be investigated, and proposes methods to use the knowledge learned by the internal neurons also for tasks different from those the network was originally trained on. In Chapter 6 an evolutionary approach is used to generate adversarial source code instances able to deceive a state of the art neural network trained in the detection of software vulnerabilities. The study of the internal behaviour of the neural networks is also investigated in the two

next chapters, but from different perspectives: in Chapter 7, by means of a mutation-based evolutionary algorithm the decision process of the network is explored by studying how the decision changes when mutations involving different syntactical elements are applied, while in Chapter 8, by considering the input instances as points in the high-dimensional vector space yielded by the activations of the internal neurons in specific regions (e.g. layers), the interest is moved in searching evidence of how human-defined concepts affect the final prediction.

Finally, in the closing part, Chapter 9 resumes and discusses the results obtained, and provides some insights for possible further research directions.

2

LITERATURE OVERVIEW

This thesis in set into a broad research area, that draws on a wide range of varied topics, from that of source code analysis by means of machine learning models, to the endeavour of explaining some of the internal dynamics and operation of such models, even with the aid of evolutionary techniques based on formal grammars. This chapter provides a summary of the scientific works related to the scenario in which the presented studies will roam, with the intention to both introduce the reader to the topics involved, and to present the main results and approaches that are available in the scientific scene. In particular, an overview of how machine learning has been utilised in the field of source code analysis will be tabled, with a focus on vulnerability detection and program comprehension. Then, some evolutionary approaches built around the use of formal grammars will be showcased, specifically when designed to synthesise programs written in high-level languages (some technical details will be thereto given in Chapter 3.2). Finally, an outlook on the field of explainable artificial intelligence will be featured, with an interest on the methods that will be devised in the thesis.

2.1 MACHINE LEARNING FOR SOURCE CODE ANALYSIS

Chasing the success achieved in a wide range of domains, such as those of images and NLP, systems based on machine learning are becoming popular also for dealing with source code (see e.g. [80] and [4]) and, more in general, with software artifacts [38]. To this end, the recent literature features several examples of ML models trained in solving tasks related to the source code processing domain, including code completion [22, 81, 89], code summarization [6, 141, 142], and classification [15, 52, 102]. The choice of the input representation for feeding such models is, in general, a crucial aspect in this scope, since it is not always effective to use the pure textual representation as in classical NLP models. It is common that more structured program encodings, such as the abstract syntax tree, the dataflow graph [69], the call graph [124] or the control-flow graph [7] are used for these kind of applications. Besides the aforementioned standard graph representations, there also exist attempts to study more powerful artifacts. Classical examples are the vector representations, known as *program embeddings*, that will be presented with more details in the next Section 2.1.1, but also other kinds of structures are possible. To this end, several works are focused in the design of program encodings or representations that are able to capture different properties, to properly convey the seized information and to help in the solution of a specific task. Interesting approaches in this direction are, for instance, the work described in [5], where a graph-based representation for pro-

grams derived from the abstract syntax tree (AST) is used for solving classical software engineering tasks such as predicting the name of a variable or if a variable has been misused, and that of [155], where a novel AST-based neural network is proven effective to address the problem of clone detection and software classification.

Recently, besides the use of networks that need as input specific program representations, also the models commonly known as transformers [140], widely used for NLP applications, are becoming popular in the source code processing domain [2, 28, 48, 65]. There are many advantages in using such kind of models. The two most evident are the convenience in the preprocessing, since the source code needs only a simple standard tokenization, and the fact that they only need to be pre-trained (in a self-supervised way) once on large corpora of data, and then fine-tuned to address specific supervised tasks. The benefit resides in the fact that, even though the pre-training phase usually requires many days and a large supply of computational resources, the fine-tuning is computationally simple, and this allows to easily operate on pre-trained models that are available in public repositories, e.g. CuBERT[1], CodeBERT[2] and PLBART[3]. More technical details on transformers for source code, along with a discussion on some of the limitations that they present will be given in Chapter 3.1.

2.1.1 *Source Code Embedding*

Particular emphasis in this thesis is given to vector representations of programs, namely on source code embedding, since many state-of-the-art neural models require a preprocessing that, in some way, maps the source code into a vector space, and then such vectors are used to feed the models and to address the deemed problem. The core idea behind these kind of approaches is the same that underlies their NLP counterparts (i.e. document embedding), that is to produce something similar to a semantic hashing [125] for source code, meaning a system that maps *similar* code snippets into nearby vectors. The value of this insight is particularly clear in works such as that of [74], where the similarity of the vectors in the embedding space is used to implement a search engine for source code. A similar idea is tackled in [10], where the authors propose a vector representation for programs for predicting the name of a method. Such embedding aims to preserve the formal structure of the code using, for the training, the possible leaf-to-leaf paths on the AST [9], and labels which correspond to the names of the methods (this approach produces vectors representing methods). The main motivation in addressing the method naming task is related to the idea that, since in most cases the name of a method is given by the programmer in order to summarize its purpose, the fact of being able to predict that name corresponds somehow in being able to understand what a snippet of code does.

1 https://github.com/google-research/google-research/tree/master/cubert
2 https://github.com/microsoft/CodeBERT
3 https://github.com/wasiahmad/PLBART

In fact, all the works described above aim to associate a small and meaningful information, for example a word or a short sentence, to parts of the source code in order to capture some semantic information. A comparable intention is tackled for instance in [102], where a tree-based embedding is used to feed a convolutional neural network for classifying programs accordingly to their functionalities.

Applications of source code embedding also exist in the field of cybersecurity: in [146], for instance, the API symbols and common API usage patterns are used to define a source code embedding to assist in the discovery of known vulnerabilities in given C functions, while in [33] different embedding are used to identify correct program patches.

In this thesis, a new general purpose source code embedding designed to integrate both structural and semantic features will be proposed in Chapter 4, along with some evidence of its effectiveness in classifying programs according to the software vulnerability they exhibit. Thereto, further discussion on benefits and limitations in using program embeddings will be given in Chapter 5.

2.1.2 *Vulnerability Detection*

In the field of cybersecurity, aside from the classical static and dynamic analysis techniques for detecting malware [63] or vulnerabilities [88], in the context of this thesis the main interest lies in the approaches based on machine learning. The detection of software vulnerabilities, in particular, plays a key role in the design of secure computing systems. One of the aspects that contribute in the security of a system is the quality of the underlying program source code. For this reason, many studies have been devised to assess the reliability of the source code. Many examples can be mentioned: the embedding proposed in [146] is used to identify C functions that are vulnerable to known weaknesses; the authors of [47] propose a Long Short Term Memory network for detecting vulnerabilities in PHP programs. VulDeePecker [82] and its extension SySeVR [84] are aimed to be complete deep learning-based frameworks for identifying weaknesses in C programs. These last approaches are based on a code representation that aggregates *code gadgets*, that basically are lines of codes that are semantically related to each others, on the basis of the data flow associated to the arguments of some library function calls. Beyond the mentioned examples, the reader is referred to [83] for a comparative study and to [86] for a comprehensive survey on deep learning based methods for vulnerability detection.

Despite the growing interest in applying deep learning techniques for automated vulnerability detection, and the very good accuracies that such systems prove to have, recent research tells that their performance drastically drops when they are used in real world scenarios [26]. State of the art models, in general, suffer from issues with the training data, such as data duplication and unrealistic distribution of vulnerable classes, and also with the model choices, for instance the

simple token-based input representations that are becoming almost a standard in the very recent years, probably due to the convenience in the preprocessing phase (almost no preprocessing is needed) combined with the growing popularity of NLP models that use similar representations for the input texts [87]. As a result, these approaches often do not learn features related to the actual cause of the vulnerabilities. Instead, they learn unrelated artifacts from the dataset, such as identifiers of functions and variables or, more in general, "natural" elements that are included in the code.

For this reason, one of the key topics of this thesis will be the study of the internal states of neural networks trained to address specific source code-related issues, to eventually identify what are the features that most affect the prediction (Chapters 7 and 8) or to possibly deceive them (Chapter 6).

2.2 FORMAL GRAMMARS AND EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are a family of metaheuristic optimization algorithms [94] which draw their inspiration from the biological principle of evolution, where a *population* of *individuals* evolves through a number of *generations*, by mutating and recombining the individuals so that the *best* individuals (according to a *fitness* function, that is a measure of goodness) ideally have more chances to survive than the others. During the years many evolutionary algorithms have been designed to address many different issues (see for instance [32] for a survey) with successful results, since they basically allow to explore a solution space whose shape is not known a priori, meaning that no assumptions on the fitness landscape are needed in general, even though studies aimed at analysing such landscapes have been proposed [111].

In the context of this thesis, where the focus is in the field of source code analysis, particular concern is given to evolutionary algorithms designed to evolve individuals that represent programs. Specifically, given the fact that source code is usually written in an high-level programming language, and that a programming language is defined by a context-free formal grammar [73], the interest is naturally diverted to the grammar-based algorithms, that allow to evolve individuals that are compliant with a given formal grammar. Technical details on the two algorithms involved in the reported studies, namely GE and DSGE can be found in Chapter 3.2, while a literature overview of evolutionary techniques for evolving programs is outlined in the next section.

2.2.1 *Evolutionary Program Synthesis*

Grammatical Evolution (GE) [106, 123], first proposed in 1998, is an evolutionary algorithm that, similarly to what Genetic Programming (GP) [76] does with syntax trees, can evolve programs that comply with a given formal grammar expressed in Backus-Naur Form (BNF).

The main difference with GP lies in how the genotypes of individuals are represented: while in GP the objects to evolve are in the form of syntax trees, and thus operators like crossover and mutation operate on trees, in GE the individuals are represented as vector of integers (as detailed in Chapter 3.2), and thus crossover and mutation are defined as in standard evolutionary algorithms (EA) [13, 150]. In GE, the evolutionary process operates on genotypes in the form of vectors, and for such vectors there exist a procedure that maps it into its corresponding phenotype, namely a string that is compliant with a formal grammar [30]. Hence, for its inherent nature, GE can be applied for addressing the problem of program synthesis [35]. Although a recent work [135] illustrates some of its limitations in effectively solving general tasks, other works point out its effectiveness under certain conditions. In particular, since the process of synthesising a program is evidently connected to the features of the specific programming problem to be solved, the authors of [61] show how the knowledge of the problem domain can be used for designing a grammar that effectively features productions that enable actions useful towards the solution of the problem itself. A similar intuition is exploited in [105], where GE is used for synthesising programs that solve the classical problem of integer sorting.

Despite the insightful results obtained, GE suffered some criticism. The major featured problems are redundancy, meaning that many genotypes are mapped into the same phenotype, and low locality, which entails that a small perturbation in a genotype leads to significant changes in the corresponding phenotype in the genome representation. Locality in particular has been object of studies [25, 119], in which it is pointed out that the more a change (e.g. a mutation) is applied in the left part of the genotype, the more it is disruptive in terms of modifications that occur in the phenotype (this is discussed in Chapter 3.2), an also that mutation-based searches are not effective when applied to GE genotypes. In order to face these drawbacks, adjustments on GE have been proposed: structured grammatical evolution (SGE) [92] and its improved dynamic version (DSGE) [91] have proven to outperform GE in solving the problems on which they have been compared. This approach seems to be very promising since, as it is detailed in Chapter 3.2, the changes introduced in the genotype representation, that in DSGE becomes a list if integers vectors in which each position is referred to a non-terminal symbol in the grammar, significantly reduce the locality and redundancy issues that affect standard GE.

In this thesis, both GE and DSGE will be used, always through the application of fitness functions derived from the output or from the internal activations of an artificial neural network trained for source code processing applications. In particular, in Chapter 6 GE will be applied to look for adversarial examples to deceive a network trained in the detection of software vulnerabilities, while in Chapters 7 and 8 DSGE will aid the analysis of the elements that affect the decision process of a neural models.

2.3 EXPLAINING MACHINE LEARNING MODELS

With the great diffusion that machine learning models played in the recent years, the fields of Explainable Artificial Intelligence (XAI) [58], and of interpretable machine learning [101] are becoming growing research areas that aim to *explain* and *interpret* how machine learning models make their predictions. This thesis presents a set of studies related to the source code processing domain, always with the aid of machine learning models. For this reason, the explanation of the internal dynamics of such models is of interest in this scope. This section provides a concise literature overview of the topics involved in the works described in the next chapters, namely the search of adversarial examples for a vulnerability detection system (Chapter 6), the identification of the syntactic features to which a network is more sensitive (Chapter 7), the analysis of how the neurons can be ranked according to their ability to detect some human concepts (Chapter 5) and the study of how similar concepts emerge in the internal states of modern neural networks (Chapter 8).

2.3.1 *Classification Errors*

Machine learning models based on neural networks classify input through a non linear mapping of input instances to output class probabilities [75]. The mapping function is shaped by the trained weights of internal connections, and classification errors arise when this function has areas of input space which generate unexpected outputs.

In the literature, the goal to understand the overall input-output behavior of a trained network, or just to discover input areas where the model delivers wrong predictions, are pursued by estimating the shape either of the manifold corresponding to the learned function or of the error function. In [137], for instance, the input space of a network in the regions around known positive instances is studied to look for adversarial examples, while the authors of [60] look for adversarial example images against which a defense is harder, by searching input space for points strategically distant from decision boundaries. Also working on images as input, [67] develops a method to generate borderline instances, and also discusses how to generate instances moving far from decision boundaries to explore how the classification behavior of the network changes around that space.

2.3.2 *Analysis of Single Neurons*

Also the study of the internal behaviour of neural models is becoming popular, and many research results in this direction show how the analysis of the activation patterns that a neuron exhibits is of interest, both in terms of the internal representations it develops and when considered only for its inherent dynamics. A recent work [149], for instance, proposes a study of these dynamics from an information theoretic perspective, while in the area of image analysis an interest-

ing approach for studying the internal representations developed by the neurons is proposed in [79], where each neuron of an unsupervised trained network was evaluated with respect to a given image classification task, with insightful results. More recently, results have been obtained for evaluating single neurons for sentiment analysis tasks [114], or in networks trained to model natural languages [34].

2.3.3 *Saliency, Features and Concepts*

In many realistic scenarios, numeric scores such as accuracy and precision are not sufficient for valuing the quality of machine learning predictions [43]. Particularly, such metrics only give information on the amount of times a model gives a correct or a wrong answer, but do not provide insights on what is important for the model for making its predictions. Therefore, the literature shows several examples in the direction of identifying which features are important for a network for making its decision [45]. Input features, however, are often meaningless for a human being, whose decision processes involve concepts at an higher level.

To this end, the literature presents several works aiming at identifying which features are important for a network to make its decision [45, 104, 107]. The basic idea of such approaches, is to compute the gradient of the output with respect to the input features. This is suitable for domains such as that of image processing [132, 152], but in other fields (e.g. Natural Language Processing (NLP)), where the object to analyse is a point in a discrete space and it needs a transformation to be used as input for a neural model, it is less effective. The main problem is that, in general, such transformation from the object to a numeric input vector is not invertible, and thus operating directly on the input features is not convenient. In literature, we find an approach to this problem when instances are graphs in [157].

In this regard, identifying if human understandable concepts emerge in the internal states of a deep model (i.e. in its neural activations) is an intriguing and growing research topic. In the image processing domain, a past work [79] shows how the internal neurons of a deep autoencoder can be effectively used as classifiers for patterns not taught during the training phase (e.g. cats or faces). More recently, concept-based explainability of networks has been investigated by means of *concept activation vectors* (CAVs) [71], which have proven to be effective to model human-understandable concepts in the internal states of a network, and that have been effectively applied also in many different domains, such as that of chess [98].

2.3.4 *Adversarial Approaches*

Any machine learning result obtained by using deep neural networks has one limit: it suffers the fact that the neural network is mostly a black box, with respect to the explainability of how it is producing its output and whether it hides problematic decision points. Evidence

of this is discussed in works related to *adversarial examples*, which are input instances crafted to fool state-of-the-art neural networks. In addition, another key element is that such adversarial instances can also be only slightly different from instances which instead are correctly classified by the same machine. This is especially apparent in the field of image recognition [56], where some research results show how to find images which are strong adversarial examples, usually by examining the gradient of the cost function of the given backpropagation network [56]. Also evolutionary approaches have been applied to address the problem of finding adversarial instances. In the image processing field, for instance, in [103] the adversarial examples are generated both by gradient ascent and by using evolutionary algorithms.

Adversarial examples can be even constructed for models built for source code processing, simply by applying basic semantic-preserving perturbations to the source code, for instance by renaming some identifiers or by introducing unused variables [147, 154], or more sophisticated ones, such as by using different control flow structures or by changing the API usage [113]. In general, when looking for adversarial examples, new knowledge is gathered about the behavior of the network, and this knowledge helps us in the explanation of what is happening inside it [42].

Given the increasing role of machine learning systems in many decisions impacting the society, or just the security of many modern systems, the explainability of artificial intelligence tools is a growing research area [1, 115].

3

BACKGROUND NOTIONS

This chapter gathers some technical background notions that accompanied the work carried out during the development of this thesis. Then, it is not intended as a complete manual, but a collection of information that can ease and enhance the reading of the core chapters. In particular, the upcoming sections report the details that define the evolutionary algorithms involved in the discussion, namely GE (Chapter 6) and DSGE (Chapters 7 and 8), and the CuBERT transformer for source code (Chapters 7 and 8).

3.1 TRANSFORMERS FOR SOURCE CODE PROCESSING

Neural transformers [140] are deep learning models that have been originally designed in 2017 to deal with sequential data (e.g. natural language) but that in recent years gained popularity also in different domains, such as that of image understanding [70] and, notably for the purposes of this thesis, that of source code processing [2, 48, 65].

The architecture of a transformer, which is reported in Figure 1 consists in an encoder-decoder structure, in which the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ into a sequence of continuous representations $z = (z_1, \ldots, z_n)$, and the decoder generates from z an output sequence $(y_1, \ldots, y_m)$ of symbols, one element at a time. The encoder is composed of a stack of N identical layers, each having two sub-layers: one equipped with a multi-head self-attention mechanism, and the other consisting in a simple feed-forward layer. The symmetric decoder is in turn composed of a stack of N layers, each with the same two sub-layers of the encoder, and a third sub-layer, which performs multi-head attention over the output of the encoder stack. The attention mechanism in the encoder layers, intuitively, ensures that, for each part of the input, the produced embedding contains information about which parts of the inputs are relevant to each other, and similarly, in the decoder layers the attention draws the same information information from the outputs of previous decoders.

The benchmark model considered in this thesis, namely the CuBERT transformer [65] released by Google Research[1] is a version of the popular BERT transformer model [39] for NLP, specifically designed for code understanding purposes. In BERT, the self-supervised pretraining is performed on two tasks:

MLM the *Masked Language Model* task, which consists in predicting the correct token in a text where some words have been replaced with a special token (e.g. `[MASK]`);

1 Available: https://github.com/google-research/google-research/tree/master/cubert

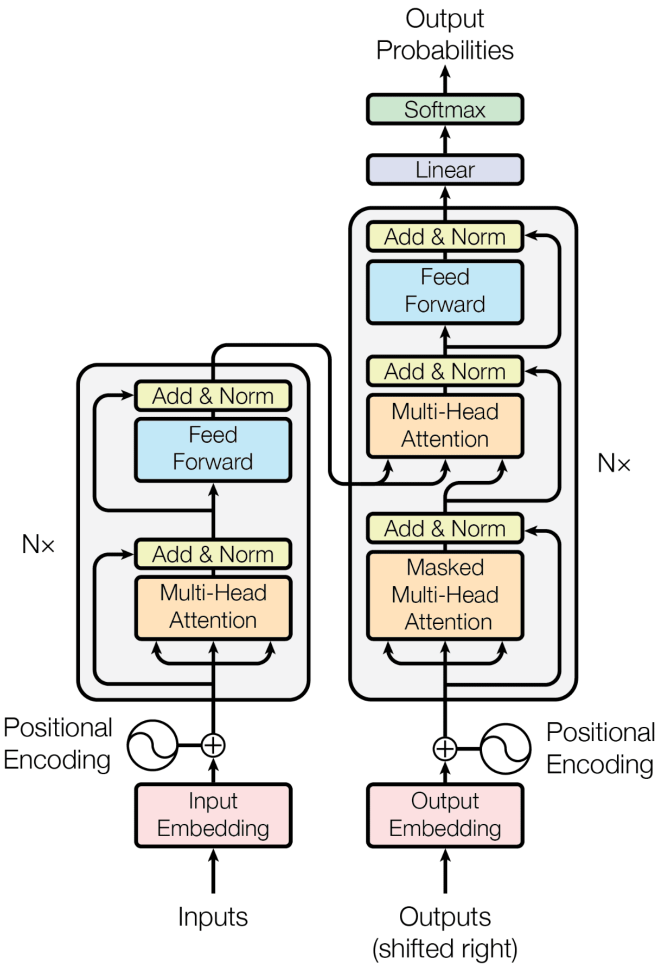


FIGURE 1: *Structure of the transformer architecture. Image taken from the paper "Attention is All You Need" [140].*

NSP the *Next Sentence Prediction* task, that consists in predicting if two sentences separated by a special token (e.g. `[SEP]`) follow each other in some point of the training corpus.

CuBERT has the same structure of BERT-large, with 24 layers, each having 16 attention heads and 1024 hidden units. The pre-trained model (for the Python language) is trained on the MLM and NSP tasks, with the caveat of considering as a sentence a *logical* line of code, namely the shortest sequence of consecutive lines that constitutes a legal statement. The authors of CuBERT have made available, in addition to the pre-trained model, six further models, each fine-tuned on on a different supervised task related to program comprehension:

1. FUNCTION-DOCSTRING MISMATCH: a sentence-pair classification problem where the function and its docstring form a pair of sentences. The positive examples come from the correct function-docstring pairs, while negative examples are obtained by replacing correct docstrings with docstrings of other functions;

2. EXCEPTION CLASSIFICATION: multi-class classification problem consisting in the prediction of the correct exception type (among the 20 most common) raised by a function;

3. VARIABLE-MISUSE CLASSIFICATION: the same classification problem addressed in [139], that consists in predicting whether there is a variable misuse at any location in a given function. To create the buggy examples, in some points of the dataset a variable has been replaced with another one defined within the function;

4. SWAPPED-OPERAND CLASSIFICATION: similar to the variable-misuse task, but it consists in detecting if operands of non-commutative binary operators are swapped;

5. WRONG-BINARY-OPERATOR CLASSIFICATION: originally proposed in [112], consists in detecting whether a binary operator in a given expression is correct. The negative examples are created by randomly replacing some binary operator with another type-compatible operator;

6. VARIABLE-MISUSE LOCALIZATION AND REPAIR: given a function, the task is to predict one pointer to identify a variable-misuse location, and another one to identify a variable from the same function that is the right one to use at the faulty location.

The pre-training has been made by using all the `.py` files available in the public GitHub repositories of Google's BigQuery platform[2], which resulted, after having removed duplicated or similar files by following the procedure described in [3], in a dataset of 7.4 millions Python files; the fine-tunings are rather performed on the *ETH Py150* corpus [116], consisting of 150K Python files from GitHub that, after a preprocessing made as above, resulted in about 125K files.

In the work presented in this thesis, some of the described fine-tuned models will be used in Chapter 7 to study if the syntactical features to which a network is most sensitive depend to the underlying task upon which the model is trained, and if such sensitivity is the same that a human expert would expect. In Chapter 8, instead, a pre-trained model will be fine-tuned in detecting different kinds of software vulnerabilities.

3.2 GRAMMAR-BASED EVOLUTIONARY ALGORITHMS

Evolutionary computation, and in general population-based optimization methods, take inspiration from the natural principle of evolution: a population evolves through many generations by recombining the genomes of the individuals (that, sometimes, can also experience some small mutations), and the best genes, according to some criterion, survive along the generations. For a complete overview of these kind of optimization techniques, the reader is referred to [94].

2 https://cloud.google.com/bigquery

Genetic algorithms (GA), invented by John Holland [62] and genetic programming (GP), invented by John Koza [76], in particular, can be roughly described by the following procedure:

1. randomly generate an initial *population* and assess the *fitness* of each *individual*;
2. by applying *crossover* and *mutation* to the individuals, create a new generation, and assess the fitness of the new individuals;
3. repeat point 2 until an ideal solution is found, or the maximum number of generations is reached.

What they differ in is how the individuals (or *genomes*) are represented: while in GP individuals are syntax trees, in GA they consist in numeric vectors. In this dissertation, we will focus on GA, since the used algorithms, namely GE and DSGE, are derived in their founding ideas from GP, but act with genotypes and operators similar to that of GA. The terminology that will be used in this section and in the upcoming chapters is the following:

FITNESS FUNCTION a measure of the quality of an individual

INDIVIDUAL a candidate solution

GENOTYPE representation of an individual during the evolutionary process

PHENOTYPE the actual representation of the candidate solution

GENE a position in the genotype

CROSSOVER an operator to recombine two PARENT individual and obtain two CHILDREN

SELECTION a method (based on the fitness) to select the parents

MUTATION an eventual alteration of a gene in a child, can occur after the crossover

GENERATION a complete cycle of fitness assessment, selection, crossover and mutation

Starting from the outlined notation and high-level algorithmic overview, we now provide the description of two population-based algorithms that allow to evolve individuals whose genotypes can be decoded into phenotypes represented by strings that are compliant with a given formal grammar.

3.2.1 *Grammatical Evolution*

Grammatical evolution (GE), first proposed in [106], can evolve programs in an arbitrary language by using an algorithmic workflow similar to that of GA, and by providing a procedure to map a genotype into the corresponding phenotype by means of the production rules

defined by a context-free grammar (CFG) expressed in Backus-Naur form (BNF) [14], that consists in terminal symbols, that can appear in the language, and non-terminal symbols, which can be expanded into one or more terminals and non-terminals. Formally, a grammar G can be expressed as a tuple $G = (N, T, A, P)$, where N and T are the sets of terminal symbols and non-terminal symbols, respectively, $A \in N$ is the axiom, or the starting symbol of G and P is the set of production rules, that define how each non-terminal can be expanded in the corresponding sequences of symbols. In the example reported in Figure 2a, for instance, we have N = {<expr>, <op>, <var>}, T = {+, −, ∗, :, x, y, z}, and A =<expr>, and P = {Rule1, Rule2, Rule3}. GE can now proceed as a standard GA [62], where the genotypes can be mapped into the corresponding phenotypes by applying from left to right the following function:

$$r = c_i \pmod{n_r}$$

where $r \in P$, n_r is the total number of possible expansions that can be applied to the non-terminal symbol associated to the rule r, and c_i is the i-th gene of the genotype being decoded. An full example of the mapping process is reported in Figure 2 where given a BNF grammar, the genotype represented by the sequence

14	7	5	12	3	9

into the corresponding phenotype

z+x

The other evolutionary operators and parameters are the same that can be applied to standard GA [94]. Notice that, with this mapping definition, the issues of redundancy and low locality of GE already mentioned in Chapter 2.2 are now evident: the redundancy is due to the use of modular operations. Moreover, since the genotype-to-phenotype mapping is performed from left to right, not necessarily all the genes in the right part of the genotype will be involved, since if at a certain point all the non-terminals are resolved, the decoding process ends and all the remaining genes are ignored. The problem of low locality, instead, is related to the fact that even a small perturbation in the genotype, for instance a mutation that increment or decrement of 1 a single gene, can drastically modify the phenotype, just think to the case of a mutation occurring in the first gene, always associated to the axiom of the grammar: in that extreme case, the phenotype would be completely different.

In this thesis, GE will be used with insightful results in Chapter 6 to automatically generate C code snippets to be inserted in other functions with semantic preserving transformation, with the aim to define a method to automatically generate adversarial input instances for a source code classifier.

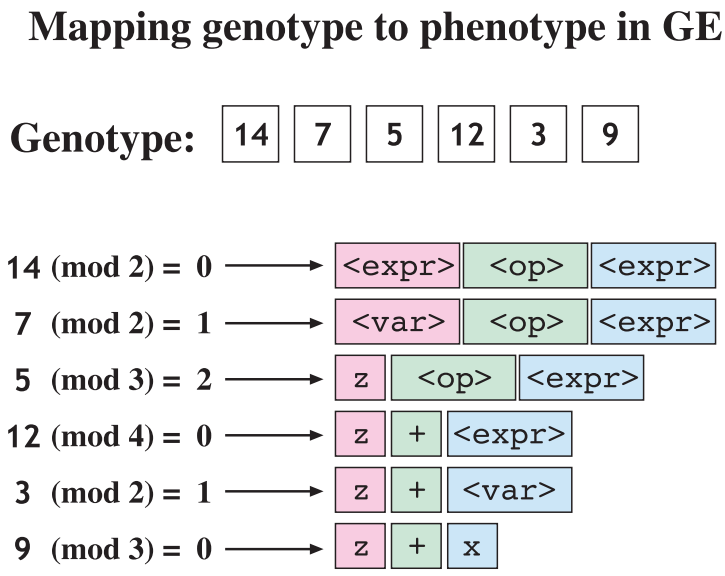
Grammar:

```
Rule 1 :  <expr>  ::= <expr> <op> <expr>   (0)
                  |   <var>                (1)

Rule 2 :  <op>    ::= +                    (0)
                  |   -                    (1)
                  |   *                    (2)
                  |   :                    (3)

Rule 3 :  <var>   ::= x                    (0)
                  |   y                    (1)
                  |   z                    (2)
```

(A) *Example BNF grammar.*



(B) *Genotype to phenotype mapping process.*

FIGURE 2: *Example of the decoding process* (B) *of a genotype into the corresponding phenotype in GE, according to the given BNF grammar* (A).

3.2.2 *Dynamic Structured Grammatical Evolution*

Structured Grammatical Evolution (SGE) [93], is a grammar-based algorithm derived from GE, which aims to overcome some of the GE issues discussed in the previous section. It operates like GE, and the difference lies in how the genotypes are represented – vectors of lists of integers – so that each gene represents a specific non-terminal, and each integer is referred to a possible expansion option for that non-terminal, as reported in the example in Figure 3. Actually, SGE requires the grammar to be pre-processed to remove recursion. For this reason, a dynamic version called DSGE [91] has been introduced to address this issue.

Given a grammar $G = (N, T, A, P)$ defined as in the previous section, each genotype is represented as a sequence $S_1, \ldots, S_n$, with $n = |N|$. Each S_i is an ordered list of integers $r_1, \ldots, r_k$, where k is

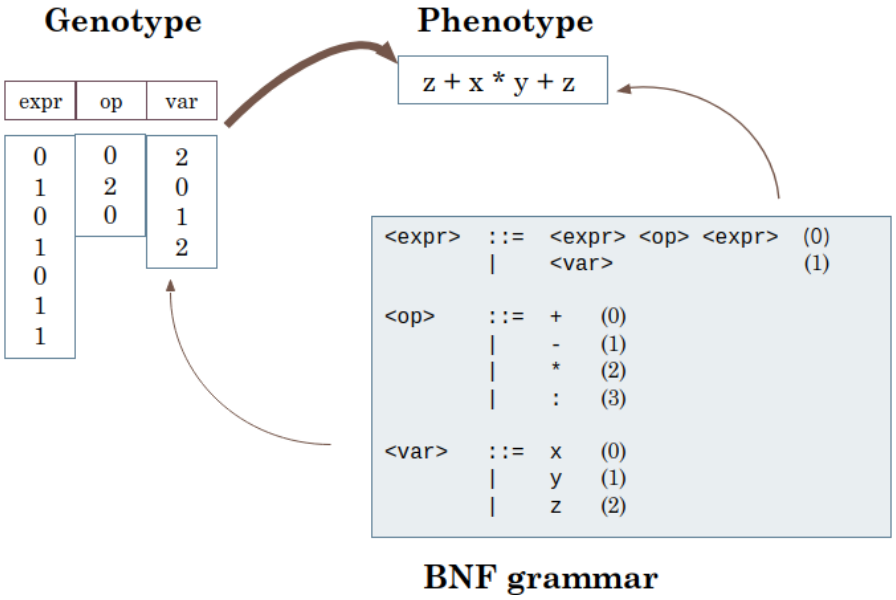


FIGURE 3: *Genotype representation in DSGE.*

the number of times the i-th non-terminal is expanded, and the values correspond to the indices (in the grammar) of the applied expansion rules. The mapping of a genotype to the corresponding phenotype, starts from the first number of the list S_1, which is always referred to the axiom of the grammar, and then proceeds by expanding the non-terminal symbols, from left to right, as in the example showed in Figure 4. For what concerns the genetic operators, as detailed in [93], mutation consists in changing an integer in a list S_i with another random one whose value is between 1 and the number of derivation options available for the i-th non-terminal, while crossover is performed by generating a random binary mask and by recombining the genes of two parents according to this mask, without changing the lists in the genes. In this thesis, DSGE will be customized with new mutation operators in Chapter 7, and used to analyse the syntactical elements to which a network is most sensitive, while in Chapter 8 similar analysis will be performed, considering however human-defined concepts instead of syntactic features.

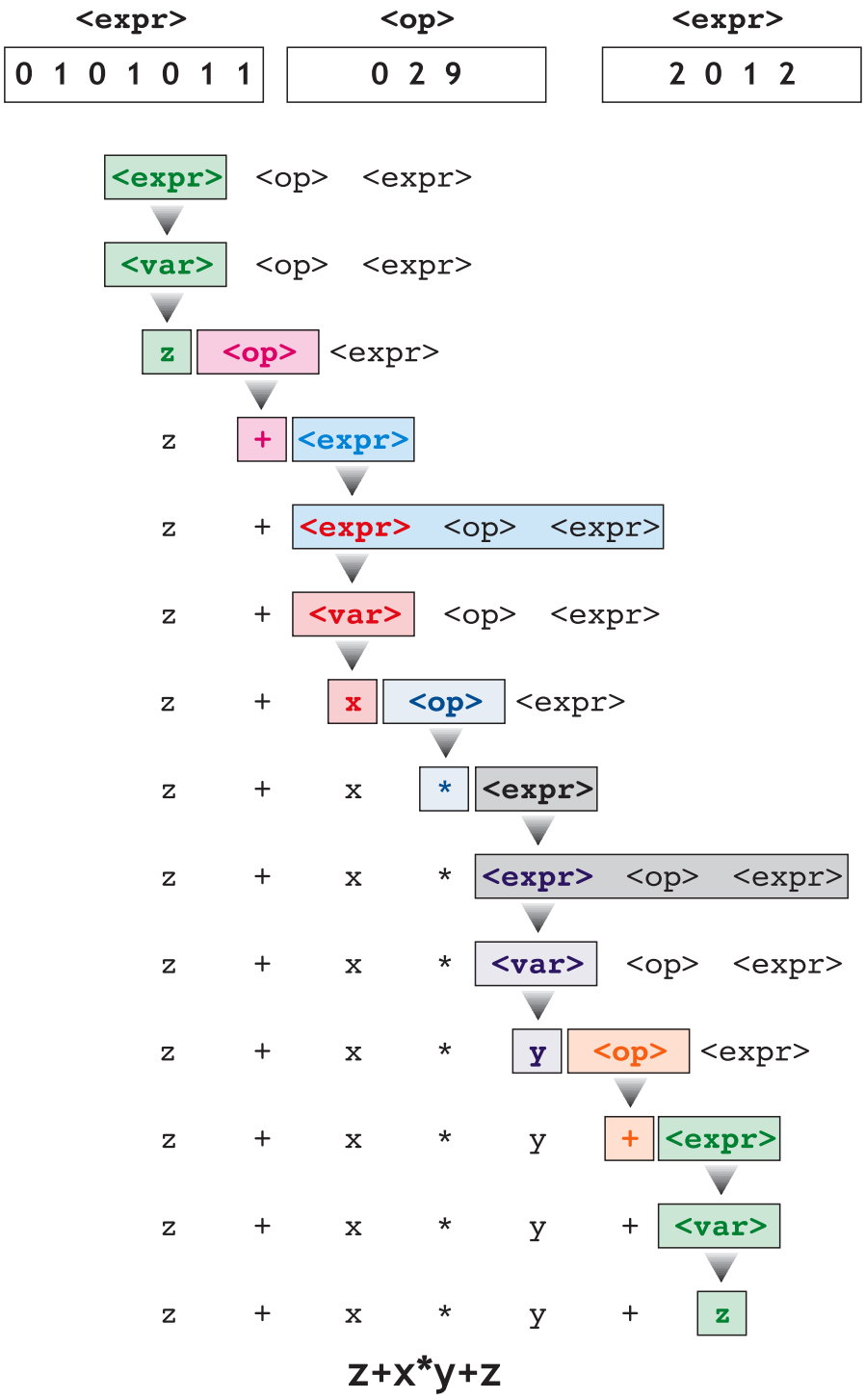


FIGURE 4: *Example of genotype to phenotype decoding in DSGE.*

Part II

RESEARCH

4

A NEURAL SOURCE CODE EMBEDDING

This chapter proposes the definition of a source code embedding, that is a technique for mapping source code snippets into a vector space. Besides, it shows its effectiveness in capturing program properties that are useful when solving security analysis tasks. The ideas, experiments and results described in this chapter have been previously published in [126].

4.1 INTRODUCTION AND MOTIVATIONS

In NLP, there exist many techniques for producing representations of words and phrases consisting in vectors of real numbers. Some of these language modeling approaches, known as *word embedding*, aim to seize some *semantic* properties of the object to be represented [27]. Due to the effectiveness that such models revealed to have in solving several and different NLP tasks, similar ideas have been spreading also in the field of source code analysis [80]. In particular, as discussed in Chapter 2.1.1, many works proposed vector representations of source code to address specific issues: for instance, the code2vec model [10] was designed to cope with software engineering problems such as predicting the name of a method; another AST-based embedding [102] attempted to classify programs according to their functionality; and the referenced names and types of C functions and common API usage patterns has been used [146] to detect software that is potentially vulnerable.

The work described in this chapter lies at higher abstraction level, since it consists in a technique for producing a general-purpose source code embedding which could assist whoever works with source code (e.g. software developers, students, researchers) in solving various tasks. As it will be detailed in the rest of this chapter, the results have been satisfactory, since the embedding showed not only its ability to map programs in vectors that preserve some structural and semantic similarities (i.e. similar programs are mapped to nearby vectors), but also its effectiveness in solving the concrete task of automatically detecting and classifying software vulnerabilities. In summary, the contribution and achieved results are the following:

- the design of a source code embedding technique based on the information stored in the abstract syntax tree;
- the investigation of the vectors' distribution in the space, that prompted how similar programs are often mapped in specific regions;
- the successful application of the yielded vectors to supervised learning tasks.

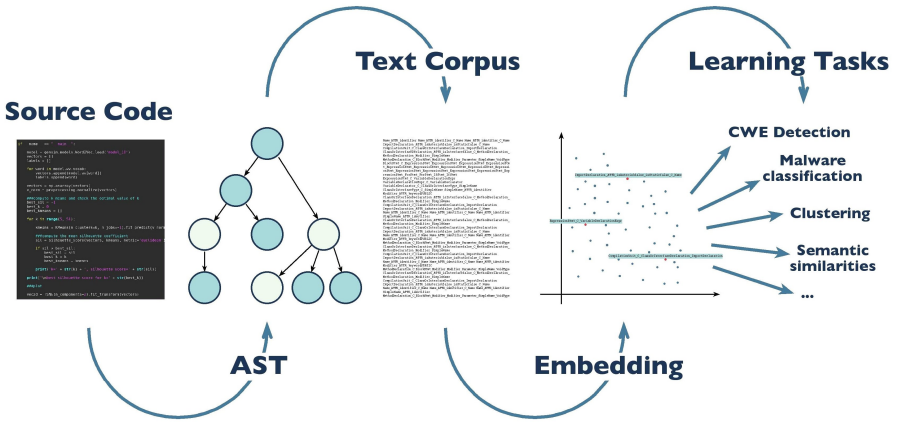


FIGURE 5: *Schema of the approach. Starting from the source code, the AST is used for producing a vector representation of programs. The obtained program embedding can be used for solving different learning tasks.*

4.2 EMBEDDING DESIGN

The main inspiration for the source code embedding design that is going to be outlined in this chapter comes from [100], in which, in a NLP context, the authors propose two algorithms (i.e. Continuous Bag of Words and Skip-gram) for producing a word embedding. Such method, commonly referred to as word2vec, is based on the *distributional hypothesis*, namely the assumption that words that often have a similar context (i.e. that often are used, in natural language, close to the same words) also have a similar meaning [121]. Accordingly to this assumption, a simple two-layer artificial neural network is trained for producing a vector space in which each word in the vocabulary is mapped into a vector, and words with a similar meaning are associated to nearby vectors. Combining the results obtained by the word2vec model and the principles through which in [125] documents are mapped to memory addresses so as to obtain a *semantic hashing*, a subsequent work [78] shows how it is possible to modify the model for producing embedding of entire sentences or documents. The results of this work are gathered in a model called doc2vec. An intriguing property that arises in both these works is that, as well as similarities among words, in the yielded vector space also more complex relations like "Paris is to France as Rome is to Italy" are preserved.

The source code embedding approach proposed here aims to exploit ideas used in the aforementioned word2vec and doc2vec NLP models for producing a vector representation which is able to capture some semantic information of the programs. The embedding is produced starting from the AST of a program, in a way that preserves the structural dependencies between different parts. This is essential since, differently from what happens in natural language, the structured nature of the source code leads to dependencies among parts of a program that can be very far from one another. Since in this work the same ideas used in word2vec and doc2vec models are applied, it is necessary to map the concepts that are commonly used in NLP

(e.g. words and sentences) into the objects that are considered in this context. If $\mathcal{A} = (V, E)$ is an AST, we define the following objects:

WORDS: in this context, a word is defined, for each node of the AST, as the node itself along with all its children. Formally, for each $v \in V$, the word w_v is a tuple $(v, v_1, \ldots, v_n)$ where, for each $1 \leqslant i \leqslant n$ it exists an edge $e = (v, v_i) \in E$ between v and v_i. Notice that we decided not to consider single nodes as words, this because such choice would have led to a vocabulary with a too small number of words;

SENTENCES: in word2vec, models need a *corpus* of sentences for the training, in order to apprehend the probability that a word is used close to another one. Since here the words derive from the AST of a program, the concept of neighborhood among words is related to the distance between nodes of the AST. Due to this fact, a sentence is defined as a sequence of words built starting from the nodes covered by performing a walk on the AST. More precisely, a sentence is an ordered sequence $v_1, \ldots, v_n$ in which it holds that $(v_i, v_{i+1}) \in E$ for each $1 \leqslant i \leqslant n$;

DOCUMENTS: in this approach, documents are defined as parts of programs with different granularities: a document can be a small snippet of code, a method or a function, a class or an entire source code file. In the performed experiments, a document will be a file containing Java source code.

Figure 5 summarizes the described approach: starting from the source code of a program, its tree-based representation (AST), that naturally derives from the grammar of the programming language in which the program is written, is used for producing a text file. Such text file can be then processed with a standard NLP library (e.g. Gensim [117]) in order to obtain a vector representation of that program. The resulting vector representation can be used for solving different learning tasks.

In the experiments, the proposed embedding, that from now on will be referred to as ast2vec, is tested on a dataset of Java programs on tasks related to program clustering and program classification.

4.3 EXPERIMENTAL SETTINGS

This section describes the carried out experiments. Starting from the preprocessing phase, in which Java source code is parsed in order to produce a text suitable for NLP analysis, in the training phase a model able to produce an embedding for Java programs is obtained. Finally, in the last phase, the model is used for inferring vectors of programs containing different kinds of vulnerabilities. The work was easily performed on a Linux machine with 4 Intel Core i7 CPUs @ 3.60GHz and 16GB of RAM.

4.3.1 *Preprocessing*

The preprocessing phase is intended to transform a Java source file into a text file. This step is implemented using the tools provided by the library JavaParser [134]. The designed pipeline is able to convert a Java source file in a text file to be used as input for a generic NLP library. The different stages that such tool performs to represent a program as simple text are the following:

1. Build the AST of a program
2. For each leaf in the tree, extract a random walk on the AST that starts in that leaf and ends in another leaf
3. For each random walk on the AST, build a list of space-separated *words*
4. Output a text file containing, in each row, a *sentence* consisting in a list of words representing a path on the AST

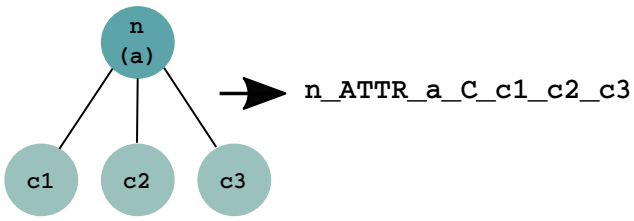


FIGURE 6: *Example of word built starting from a node* n.

Notice that, in this approach, a word is composed of a node n along with all the children of n. For this reason, the construction of a word is done by chaining the label of a node and the labels if its children, using the "_" character as a separator. Moreover, since in JavaParser some information in the nodes are stored as attributes, the tags `ATTR` and `C` are used for separating attributes and children, respectively. Figure 6 shows an example of a simple part of a tree and the construction of the corresponding word.

The simple use of all the labels and attributes of the nodes would yield a vocabulary with an unbounded number of words. This is due to the fact that possible values and identifiers that can appear in the source code are potentially unlimited. This fact could lead to problems related to the performance and to memory usage. In order to address these issues, different data preprocessing have been performed, each of them in compliance with a different design option:

IDENTIFIERS: to handle the identifiers, two different alternatives have been tested:

- Not to consider identifiers: if an identifier appears as attribute of a node, only the tag `identifier` is written in the corresponding word.

- To consider all the identifiers as they are in the source code: if an identifier `ID` appears as attribute of a node, the tag `identifierID` is written in the corresponding word.

VALUES: similarly, two alternatives have been tested for managing the values:

- To consider all the possible values. In this case, if the value 2907 appears as an attribute of a node, the tag `value2907` is written in the word.
- To consider only a small set of possible values. In this case, only the values belonging to the set $\{0, 1, \text{true}, \text{false}\}$ are considered. In all the other cases they are mapped d to the generic value `X`. In this case, the value 2907 used in the example above is written as `valueX` in the corresponding word.

Experiments have been performed by considering all the possible combinations of these design options. The results obtained will be presented in Section 4.4.

4.3.2 *Training*

The ast2vec models have been trained on the same dataset Java-small dataset used in [6], which contains some of the most popular Java projects on GitHub, according to their number of watchers, stars and forks. The dataset has been preprocessed to produce a corpus of text files that it could be used as input for a document embedding model. The gensim framework [117], provides an implementation of the doc2vec algorithm, and starting from the preprocessed files, a doc2vec model has been trained to produce source code vectors of three different sizes: 10, 100 and 300. By considering all the possible preprocessing options and dimensionality of vectors, the trainings resulted in 12 different ast2vec models. Table 1 summarizes the features of each trained model.

4.3.3 *Inference*

The last experimental phase consisted in inferring vectors of programs containing common known vulnerabilities, by means of the trained ast2vec models. The inferred vectors came from the *Juliet Test Suite v1.3 for Java* [17, 18], which is developed by the NSA specifically for testing static analysis tools. In this dataset, programs are labeled accordingly to the Common Weakness Enumeration (CWE)[1] list.

In the experiments, in order to being able to operate a more accurate qualitative analysis of the results, only the test cases that covered with sufficient number of examples of one of the CWE categories listed in the 2011 CWE/SANS Top 25 Most Dangerous Software Errors have been considered. Table 2 shows the number of examples available for each CWE category.

1 https://cwe.mitre.org/index.html

TABLE 1: *Properties of the trained ast2vec models*

Model ID	Size	IDs	Values	Memory
00-10	10	No	Subset	11M
00-100	100	No	Subset	52M
00-300	300	No	Subset	118M
01-10	10	No	All	11M
01-100	100	No	All	55M
01-300	300	No	All	125M
10-10	10	Yes	Subset	75M
10-100	100	Yes	Subset	306M
10-300	300	Yes	Subset	815M
11-10	10	Yes	All	76M
11-100	100	Yes	All	307M
11-300	300	Yes	All	819M

TABLE 2: *Summary of the CWE categories as represented in the Juliet dataset*

Rank	Description	CWE Entries	#Examples
1	SQL injection	89	2220
2	OS command injection	78	444
4	Cross-site scripting (XSS)	79, 80, 81, 83	1332
7	Use of hard-coded credentials	798, 259, 321	(148)
8	Missing encryption	311, 315, 319	407
13	Path traversal	22, 23, 36	888
19	Broken cryptographic algorithm	327	(34)
22	Open redirect	601	333
23	Uncontrolled format string	134	666
24	Integer overflow/underflow	190, 191	4255
25	One-way-hash without a salt	759	(17)

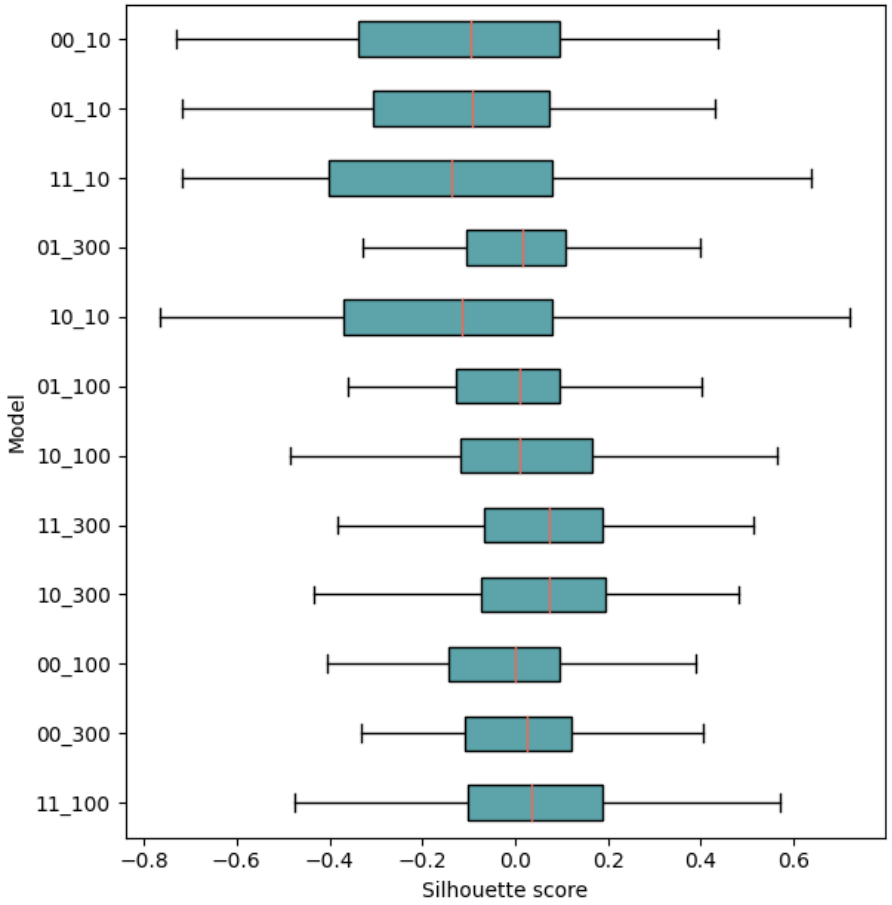


FIGURE 7: *Overall silhouette score distributions obtained by the different models.*

For each program representing one of the covered CWE categories, the corresponding vector has been inferred by using each of the trained models. The aim of this step is to figure out if the vector representation obtained with ast2vec is able to distinguish among different categories of software vulnerabilities. Since different parameters combinations have been tested, to obtain a numeric score for the quality of each model accordingly to the task, a measure related to cluster analysis have been adopted. The silhouette score [120], which gives a measure of the cohesion of each sample with the other samples in the same cluster compared to the separation of that sample from samples in other clusters, has been computed for each vector, accordingly to the partitioning yielded by the CWE labels.

The box plots in Figure 7 show the silhouette values distribution obtained by using different parameters combinations, while Figure 9 shows the spatial distribution of the vectors inferred by the model 10-300, as represented by the two-dimensional visualization produced with t-SNE [138], considering the cosine similarity as a metric.

4.4 EMBEDDING EVALUATION

This section describes the experiments performed to evaluate the properties of the proposed code embedding, with a focus on its abil-

TABLE 3: *Silhouette score obtained by different models*

Model	Mean	Median
00-10	−0.129	−0.095
00-100	−0.012	0.001
00-300	0.011	0.024
01-10	−0.127	−0.092
01-100	−0.006	0.008
01-300	0.013	0.016
10-10	−0.137	−0.114
10-100	0.012	0.011
10-300	0.057	0.073
11-10	−0.125	−0.137
11-100	0.032	0.034
11-300	0.061	0.074

ity in catching security flaws. Three kinds of evaluations have been done: a statistical analysis of how the embedding is able to naturally separate programs containing different weaknesses, a qualitative assessment of the way in which the model places programs in the vector space and the application of different supervised learning algorithms on the inferred vectors to measure the accuracy in classifying the CWEs.

4.4.1 *Cluster Analysis*

Figure 7 and Table 3 show in terms of silhouette score the performance of each embedding model in spatially separating different weaknesses, which assigns to each sample a value ranged from -1 to 1, where high values mean that a sample is well tied in its own cluster and well separated from other clusters.

In general, the results show that the models that produce vectors having 10 dimensions aren't powerful enough for autonomously distinguishing among different classes of vulnerabilities. In fact, mean and median silhouette scores in those cases have always negative values. Best silhouette values are achieved by the models 0-300 and 11-300, whereby vectors have 300 dimensions and identifiers are all considered. Such models only differ in the management of values as discussed in Section 4.3. Since both mean and median of the two models differ by less than 0.005, the model 10-300 has been used as benchmark, in virtue of the reduction in terms of memory usage for storing the model, the training time, and the better graphical results obtained in the 2D visualization (Figure 9). All the following discussion is thus made by considering 10-300 as the reference model.

For being able to obtain a numeric measure of the CWEs that are better identified by means of the proposed embedding, the silhouette

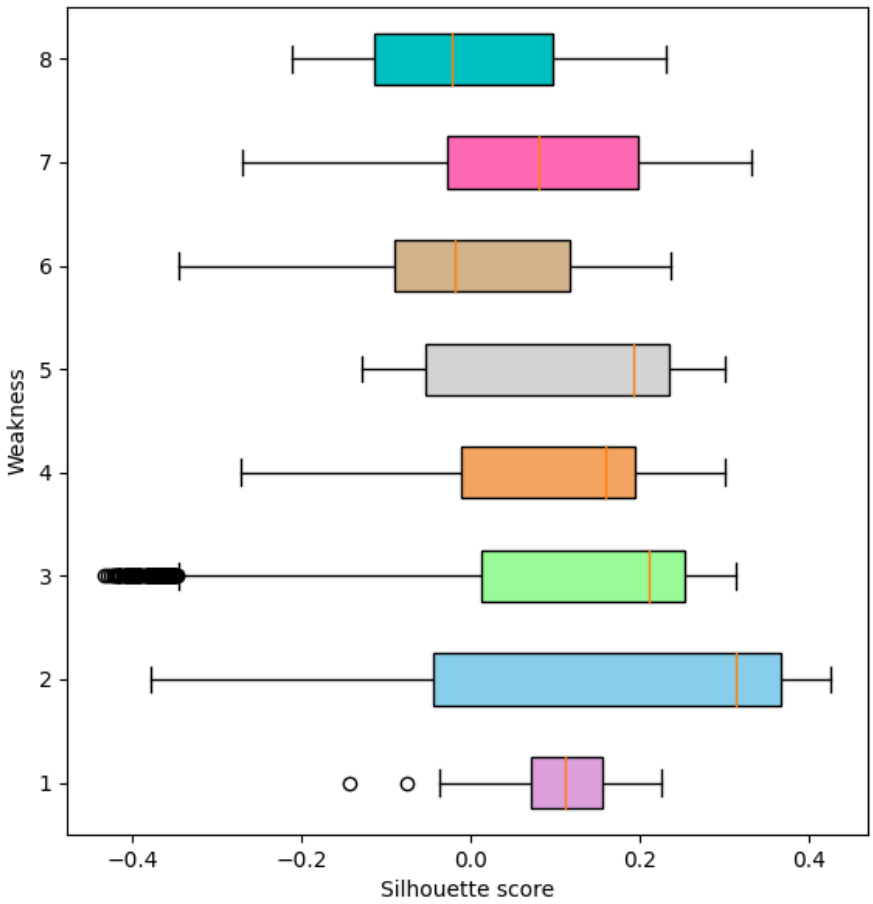


FIGURE 8: *Silhouette score distributions obtained by the model 10_300 grouped by weaknesses categories. Refer to the legend in Figure 9 for the association between colors and the different CWEs. Weaknesses with less than 150 examples in the dataset (see Table 2) have been removed for this analysis.*

coefficient of the sets of vectors inferred by the model have been computed. Results are showed in Table 4 and Figure 8. In general, the silhouette scores obtained by programs representing different weaknesses are not equally distributed among the different groups of CWEs: for instance, silhouette scores of samples in "Path traversal" are, on average, much higher than values in "Uncontrolled format string". This means, as will it be discussed below, that not all the features proper of different CWE categories are naturally seized in the same way by the vectors embedding.

TABLE 4: *Statistics of the silhouette score obtained for different CWEs*

CWE category	**Mean**	**Median**	**Variance**
XSS	0.115	0.117	0.003
Path Traversal	0.173	0.322	0.060
SQL Injection	0.104	0.207	0.044
Missing Encryption	0.097	0.143	0.015
Open Redirect	0.114	0.188	0.021
Integer Overflow/Underflow	0.009	−0.006	0.012
OS Command Injection	0.088	0.086	0.022
Uncontrolled Format String	−0.004	−0.031	0.013
Broken Cryptographic Algorithm	0.411	0.430	0.020
One Way Hash	0.256	0.282	0.021
Hard Coded Credentials	0.086	0.152	0.021

4.4.2 *Qualitative Assessment*

Figure 9 shows (in a 2D approximation) how the embedding produced with the technique described in Section 4.2 from the generic dataset of popular Java projects on GitHub placed each document of the Juliet Java dataset in the embedding space. The learning that defined the embedding is unsupervised, with no required bias toward the security features of the source code that tare associated to the instances. Nonetheless, some types of vulnerabilities can be associated to specific structures present in the source code, while others cannot. For instance, dealing with integers, an overflow/underflow can be often associated to lack of local checks on bounds, while vulnerabilities of category as "Open Redirect" (CWE-601) are related to a set of programming errors which can be far apart in source code, and finding them could require analysis of separated Java files. Therefore, the embedding developed on the generic Java data set, which has proven to capture some semantics of source code fragments, is expected to behave differently when the vectors represent vulnerable Java code labelled with respect to different CWE categories.

The following discussion analyses the obtained behavior, to show how the embedding captures security features better for some CWEs

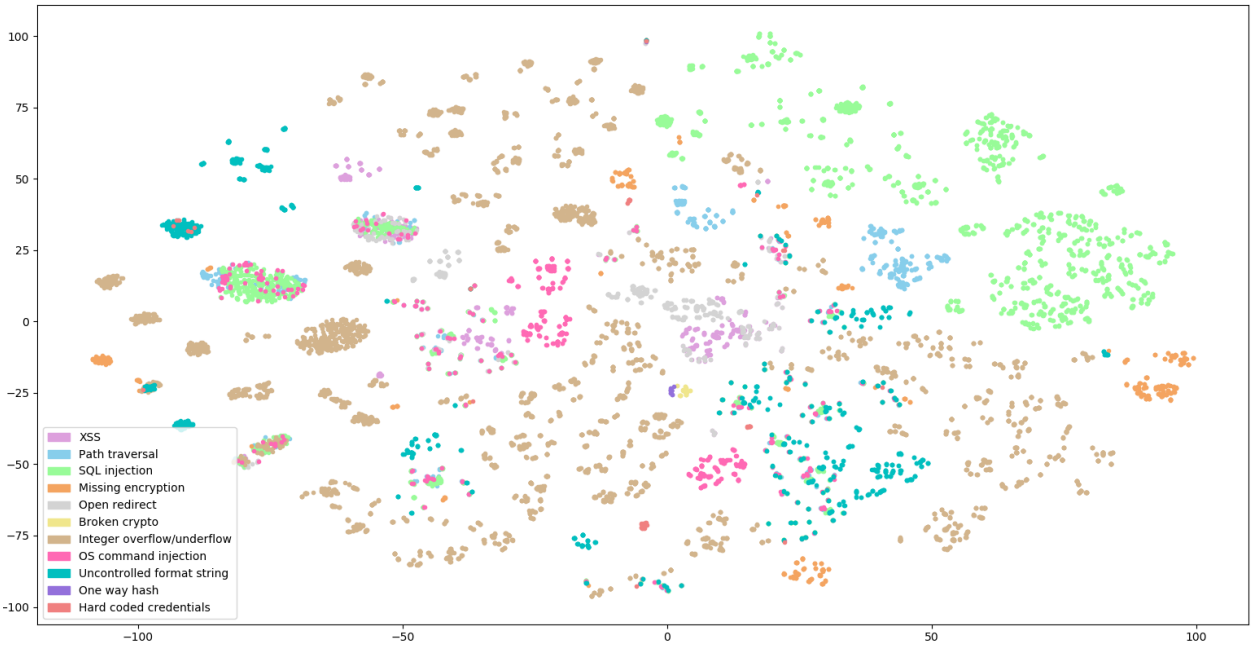


FIGURE 9: *2D visualization of vectors representing different weaknesses.*

than for others. The basic functioning of the referred vulnerabilities is outlined in Table 5. In Figure 9 each point represents a Java file of the Juliet data set, and the colors are assigned depending on the CWE category it represents. We can immediately notice that the placing of each point, naturally determined by the embedding (and by the 2-dimensional t-sne approximation), tends in general to separate vectors of different vulnerabilities. CWE such as "XSS" (violet points) and "Open redirect" (grey points) appear to behave badly; these weaknesses can reasonably be associated to bad overall engineering of software parts, and thus they cannot be detected by analysing a single file. Therefore, vectors with these labels are placed spread in the embedding space. Besides, we can notice that some colors associated to CWE categories which are mostly well separated in the embedding space, such as those of "Integer overflow/underflow" (khaki points) or of "SQL injection" (green points), in some areas are mixed with points of different colors. A manual check on these mixed points revealed that they are always associated to files that in the dataset are *base classes*, that is files that contain only simple Java classes which call other files, wherein the vulnerability is actually present. Instead, those called files are in well separated areas of the embedding space. For this reason, these base classes files have been left out from the classification experiments presented in the following section.

Finally, another observed behavior of the embedding is that the vectors belonging to a single CWE category are placed in finer clusters, each containing the points of a given CWE. This hints at the recognition of further differences between structural features of the different Java files associated to the CWE categories. The behavior of the embedding produced by means of this unsupervised learning approach, allows to devise its use to select Java files in order to help manual

TABLE 5: *Description of the discussed CWE*

Category	Description
SQL INJECTION	Code injection technique [59] in which SQL queries are sent (e.g. through a web form or a search box) from a client to a web application, to execute operations on the database, such as modify or steal sensitive data.
CROSS SITE SCRIPTING (XSS)	Injection attack [57] in which malicious code is sent (typically through a web application) to a dynamic website and then executed by the browser of a web user.
OPEN REDIRECT	This vulnerability [44] occurs when a web application allows a (malicious) user to control the redirect to another URL.
INTEGER OVERFLOW AND UNDERFLOW	It occurs when an arithmetic operation outputs a numeric value that falls outside allocated memory space or overflows (or underflows) the range of the given value of the integer [23, 40].
UNCONTROLLED FORMAT STRING	The exploit for this vulnerability [31] occurs when submitted data of an input string are evaluated as a command by the application (i.e. by means of format parameters).
UNCHECKED RETURN VALUE TO NULL POINTER DEREFERENCE	Missing check for an error after calling a function that can return with a NULL pointer if the function fails, which leads to a resultant NULL pointer dereference, e.g. through a user controlled input to a malloc() call [148].
UNCONTROLLED RESOURCE CONSUMPTION	Lack in controlling the allocation of a resource, thereby enabling an actor to influence the amount of resources consumed [11].

security assessment of a code base. Each set of clusters in Figure 9 containing points of a single color, defines a bigger cluster of vectors associated to a single CWE, and thus can be used to select files of interest for security analysis. This can be obtained by fist identifying regions of interest in the embedding space (i.e. regions associated to a given vulnerability), and then by computing the cosine similarity between the point that represents the centroid of a region and the vector inferred from the targeted file.

4.4.3 *Supervised Classification*

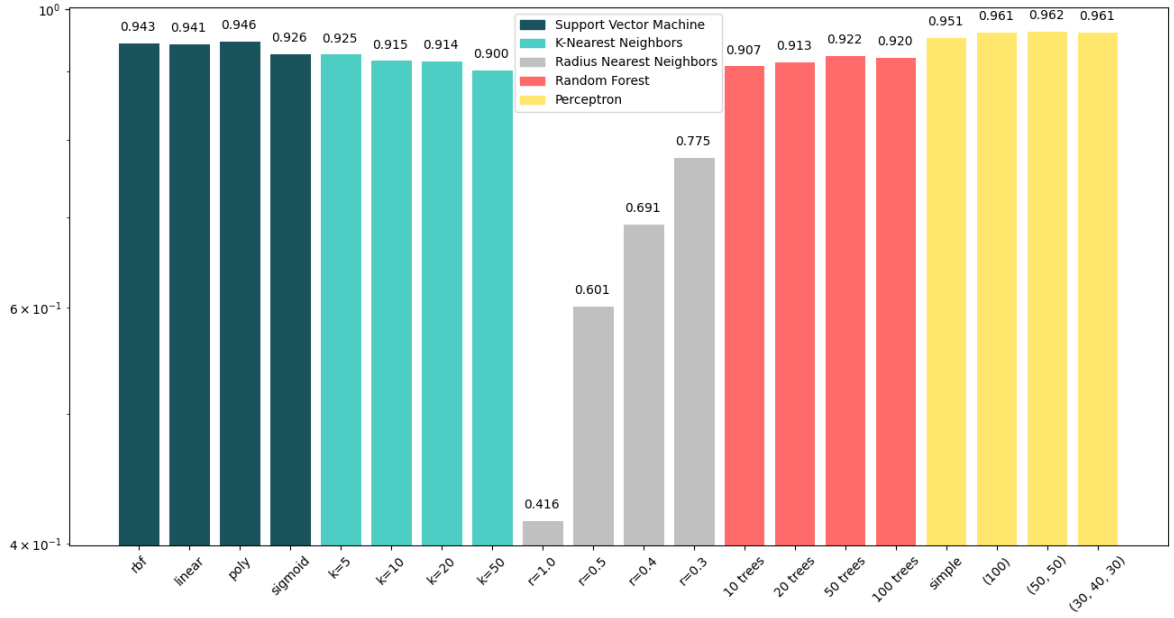


FIGURE 10: *Accuracy score obtained by different algorithms in the classification of the top 8 CWE classes from Table 4. Notation: the* i*-th element in each tuple is the number of nodes (neurons) in the* i*-th layer.*

The final experimentation aims at measure the efficiency of the embedding in being used as input for supervised learning models trained in the classification of different CWE categories. The dataset used for this test is the same subset of the Juliet test suite used for the previous discussion (see Figures 8 and 9). After a preprocessing in which all the base classes have been removed, (i.e. source files that do not directly contain any CWE, see the previous section for an explanation), only the CWE categories having at least 300 examples have been retained. The result was an 8-class classification problem.

Five different classifiers have been tested: Support Vector Machine (SVM) [19], k-Nearest Neighbors and Radius Neighbors [54], Random Forest [20] and Multilayer Perceptron [118]. A random 80-20 split of the dataset between training and test sets has been performed, and each classifier has been trained by trying different parameters: four different kernel functions for SVM, different values of k and r for k-Nearest Neighbors and Radius Neighbors, respectively, variations in the number of trees for random forest, and different numbers of hidden layers and nodes per layer in the multilayer perceptron. As reported in Figure 10, with four classifiers (out of the five tested) a

classification accuracy higher than 90% is always obtained. The best accuracy score, higher than 96%, is obtained by a multilayer perceptron with 2 hidden layers each having 50 nodes. The results suggest that the proposed embedding, produced starting from generic Java source code, together with a supervised learning model trained on vectors representing programs which provide different classes of known weaknesses, is powerful enough to capture the structural and semantic information required to distinguish among different kinds of vulnerabilities.

As a last assessment, the proposed embedding has been used for solving another multi-class classification problem. The investigation has been made on a subset of the CWEs considered in [12], in which the authors use some of the CWE examples contained in the *Juliet Test Suite for C/C++* (i.e. the analogous of the aforementioned Juliet Test Suite for Java) for solving similar classification tasks. The selected CWE entries are the same that were considered in that work and which also have a corresponding set of examples in the Java dataset, thus resulting in a 13-class classification problem. The 13 CWEs considered are listed in Table 6.

TABLE 6: *Single CWEs considered for classification*

CWE ID	**CWE Name**
CWE-23	Relative path traversal
CWE-36	Absolute path traversal
CWE-134	Use of externally-controlled format string
CWE-190	Integer overflow
CWE-191	Integer underflow
CWE-197	Numeric truncation error
CWE-369	Divide by zero
CWE-400	Uncontrolled resource consumption
CWE-476	NULL pointer dereference
CWE-563	Assignment to variable without use
CWE-606	Unchecked input for loop condition
CWE-617	Reachable assertion
CWE-690	Unchecked return value to NULL pointer dereference

The following assay analyses the ability of the embedding to better capture the features proper of single weaknesses in a set of CWEs, and also how, and why, its performance varies over them. To this end, let's consider the *confusion matrix* C yielded by the classification, in which each entry $C_{i,j}$ is equal to the number of instances in the test set that are known to be in class i and predicted to be in class j. Even if, in the previous comparison of different classification algorithms, the multilayer perceptron obtained the best accuracy score, for this last analysis SVM with polynomial kernel has been used. The reason of this choice lies in the fact that the classification performed with

the SVM yields a confusion matrix with a greater number of zeros, meaning that the errors are less scattered among the different CWEs, and thus easing and making more interesting the discussion. Finally, it is worth noting that with this setting we outperform the accuracy score of [12] in 8 CWEs out of 13.

Let's now consider the corresponding confusion matrix in Figure 11, normalized over the true labels. The high values on the main diagonal indicate that the classification is overall performed with an high level of accuracy. The worst accuracy value is the 68% resulting in the classification of the CWE-36 (Absolute path traversal), but the matrix shows that most of the errors are made by confusing CWE-36 with CWE-23 (Relative path traversal). The confusion between the two classes is also made in the inverse direction: in fact, in the 18% of cases the classifier predicts CWE-36 instead of CWE-23. These kinds of classification errors are to be expected, since CWE-36 and CWE-23 are strictly related weaknesses. A similar consideration can be made while analyzing the low accuracy (76%) obtained for CWE-191 (Integer underflow): even in this case most of the errors are made by predicting CWE-190 (Integer overflow). The last interesting outcome is that the learned model misclassifies with a label CWE-190 a number of instances from other classes, for instance with a 20% error rate for the seemingly unrelated CWE-369 (Divide by Zero), and that no other predicted label has similarly spread errors.

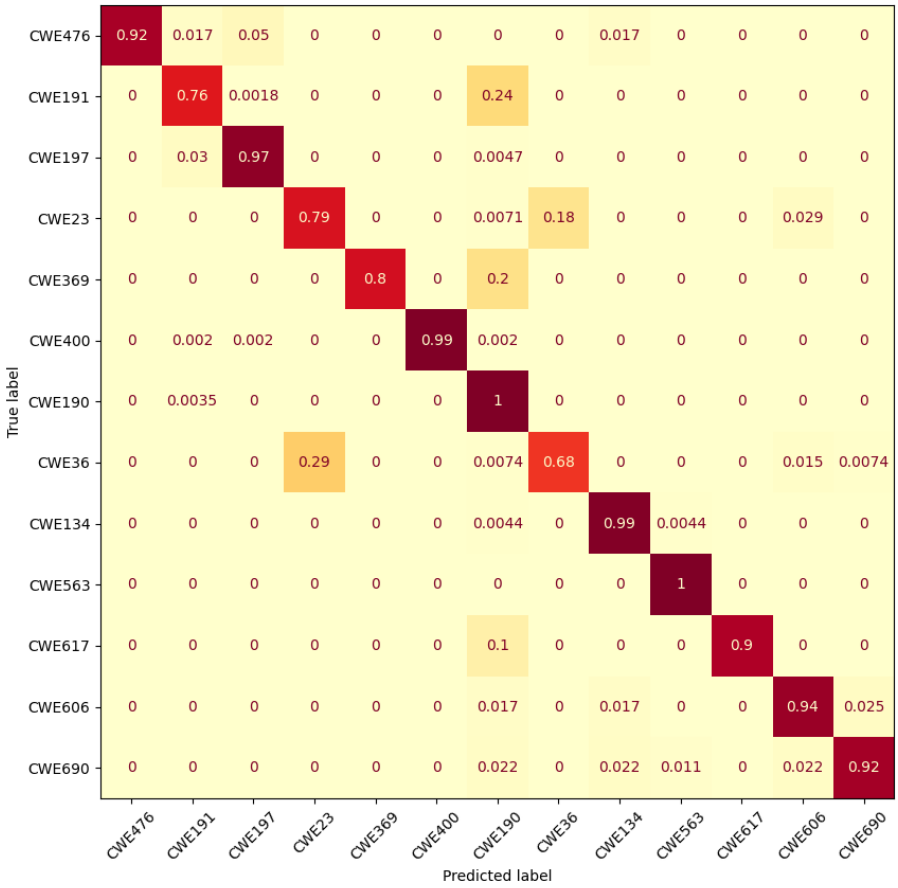


FIGURE 11: *Confusion matrix of the classification performed using SVM.*

4.5 DISCUSSION

This last section discusses the obtained results and gives some insights for future research directions.

4.5.1 *CWE Views*

A first interesting discussion point concerns the relations between the performance achieved in the classification experiments and the structured *views* that the CWE community offers of the catalogued vulnerabilities[2].

The vulnerabilities types covered by the experiments, which are listed in Table 6, are described in the CWE views as having properties which make them a varied set. For instance, two of them, namely the use of externally controlled format string (CWE-134) and the unchecked return value to null pointer dereference (CWE-690) are mentioned to be very common in programs written in C language and not common in other languages (while the other CWEs considered here are equally frequent in C as in Java). Moreover, interesting views are those listing CWEs less related to the software development lifecycle (e.g. CWE-400, uncontrolled resource consumption), or even CWEs that are not associated to software fault patterns (e.g. CWE-690). The supervised classification achieved good performance even on them, and this opens to interesting further research inquiry. In general, the use of ast2vec vectors allowed to reach good performance on these examples, even if by only considering the syntactic structure of code.

4.5.2 *Significance and Limitations*

The benchmark Juliet Test Suite has been built to evaluate tools for both static and dynamic automated vulnerability detection, and it is composed by synthetically generated examples. This can be useful to pinpoint the characteristics of tools, but it also leads to a set of examples with a simpler structure than real-life code. Moreover, the programs in the dataset often share the whole structure, and they only differ in single instructions, variable values, or very small code fragments. At a first sight, this fact could constitute a threat to validity for the assessments discussed in Section 4.4, since the natural separation among different classes of vulnerabilities resulted in the vector space and also the good classification results should be caused by the inherent similarities and not by features induced by the presence of given vulnerabilities. Actually, this is not a real concern, if we consider the whole analysis as a generic investigation on the potential of the ast2vec model, instead of a direct evaluation on its ability in classifying security flaws. In this second case, a deeper analysis should have been performed to confirm the validity of the results. As a matter of fact, the interest here was to test the ability of the ast2vec

2 https://cwe.mitre.org/data/#helpful_views

embedding to capture relevant source code properties that were of some interest in different application settings, and the results have met expectations.

4.5.3 *Further Directions*

Tu further extend this research, a natural follow-up is to test the applicability of ast2vec in real-world scenarios. As already discussed, a vector representation can be used as input for general machine learning models. A possible investigation should thus be train deep models on some supervised task, similarly to what has been done in Section 4.4, but with more sophisticated models and more reliable datasets, and to compare the performance to that of traditional or state-of-the-art tools performing the same task.

Moreover, in compliance with the NLP ideas that inspired the design of ast2vec, another interesting investigation should be to verify if semantic similarities and analogies are preserved in the embedding space. Finally, ast2vec has been applied to study the internal representations that neural models build when learning a given task. This study is thoroughly detailed in Chapter 5.

5 NEURAL DYNAMICS AND CONCEPT MINING

Deep neural networks have proven to be able to learn rich internal representations, also for features that can also be used for different purposes than those the networks are originally developed for. This chapter, whose contents have been previously discussed in a preprint [127], investigates such ability by studying the internal behaviour of networks trained for source code processing tasks.

5.1 INTRODUCTION

Research results on the use of deep learning systems show how the internal representation developed by a system during its training is of value, even for tasks different than those it was trained for. Techniques which exploit this fact are, for example, the methods for pretraining [46], transfer learning [156] or issues related to internal interpretability [53]. The research that aims at characterizing such internal representation is active for the domains of image processing [79] and of NLP [34], while fewer results are, however, available in the domain of source code processing. Given the growing diffusion of neural networks in the source code processing domain [80], similar studies deserve to be conducted also in this field.

The focus of this chapter is to examine the internal neurons of a learning system, in order to look for those which exhibit interesting behaviours, in terms of classification performance or activation patterns. The study of the internal behaviour of neural models is becoming popular, and many research results in this direction show how the analysis of the activation patterns that a neuron exhibits is of interest, both in terms of the internal representations it develops and when considered only for its inherent dynamics. A recent work [149], for instance, proposes a study of these dynamics from an information theoretic perspective, while in the area of image analysis an interesting approach for studying the internal representations developed by the neurons is proposed in [79], where each neuron of an unsupervised trained network was evaluated with respect to a given image classification task, with insightful results. More recently, results have been obtained for evaluating single neurons for sentiment analysis tasks [114], or in networks trained to model natural languages [34]. The main goal is to define a general approach for discovering and exploiting *all* the knowledge learned by a given model. For instance, one can assume to have a neural model (trained on a main task) embedded in a code editor or in a software repository that also allow to tap into further parts of the internal representation developed while being trained. To this aim, this study resulted in the following contributions:

- a procedure for ranking neurons according to their ability in solving arbitrary binary classification tasks. With the experiments in this direction some neurons revealed to be able to autonomously build internal representations for different program properties:
- an information theoretic approach for identifying neurons which exhibit interesting behaviours, with the aim to identify the most *informative* neurons in the network and to discriminate among neurons showing different activation patterns;
- a statistical measure for comparing the arbitrary binary tasks defined by single neurons (namely by simply establishing a threshold and by splitting the dataset according to the activation induced by each program instance), to identify neurons which recognise unique (or uncommon) *concepts*.

It should be noted that the use of machine learning systems entails the need of choosing in intermediate program representation to feed the model. The choice of the input representation for feeding such models is, in general, a crucial aspect in this scope, since it is not always effective to use the pure textual representation as in classical NLP models. Besides the classical structured representations (e.g. abstract syntax tree, call graph, control flow graph), also vector representations (previously treated in Chapter 2.1.1) are common and effective. In the following, we will consider as a benchmark model a neural network trained in the generic task of reconstructing program vectors.

5.2 APPROACH OVERVIEW

The proposed approach is devised at analysing the internal behaviour of neural networks trained on source code, with the aim to express its dynamics in a measurable way. Basically, it consists in the training of a simple autoencoder (i.e. an artificial neural network whose purpose is the reconstruction of the input, see [55] for a complete reference) on two different source code embeddings, and in the design and execution of three categories of experiments:

1. binary classification experiments, for ranking neurons considering their ability in solving specific tasks;
2. analysis of the relevance of the neurons for the network itself, regardless of a given task;
3. pairwise comparison of the neurons' dynamics, through the adoption of statistical techniques.

For all these experimental approaches, the first step was to map the source code to feature vectors, namely via a neural embedding, and then to study the internal behaviour of a neural network trained on

such vectors, by analysing the activation values of the neurons on different program instances. In the first two cases, the main interest is to assign a score to each neuron, i.e. in ranking the neurons according to different criteria, while the aim of the third point is to possibly define a partition for the set of neurons, in order to discriminate different behaviours and to define an association among neurons sharing similar patterns in terms of statistical distribution of the activation values.

In the classification experiments, the score assigned to each neuron is represented by the accuracy obtained when used as a classifier for given binary problems, as it will be detailed in Section 5.4. The basic idea is to consider, for each neuron, different activation thresholds and then to measure, for each threshold, the accuracy of the neuron in classifying program instances from a balanced labelled sample, when predicting a program to be in class 0 if the activation yielded by that program is less than the threshold, and to be in class 1 otherwise. The scoring mark for a neuron is the accuracy obtained while considering the threshold that leads to the highest accuracy.

In the second class of experiments, the score of each neuron is instead computed independently from any task. Similar studies, i.e. the definition of a scoring measure for evaluating the importance of single neurons in a network, have been already investigated in the literature [16, 34]. While in the referred works the core idea is to use the correlation between the activation values of neurons in distinct but isomorphic models (i.e. obtained by retraining the same model on different training sets) for finding neurons that possibly capture properties that emerge in different models, here the proposed ranking is based on the concept of *entropy* used in information theory. The reason is that, while by means of the correlation analysis one is able to state which neurons are the most important with respect to the task the network is trained on, with the proposed entropy-based measure, the neurons are graded with regard to the importance they have according to their behaviour in the network: since by definition the entropy in information theory is the average level of information emitted by a signaling system [131], computing the entropy of single neurons is equivalent to measuring how much each neuron is informative.

Finally, in the third class of experiments, all the possible pairs of neurons are considered, and for each pair, a nonparametric statistical test is performed to assess which neurons share some activation patterns. This will also allow, as it will be detailed and formalized in Section 5.5.2, to study if there exists a partition of the neurons based on their entropy levels such that (intuitively) activation patterns of the neurons belonging to the same part can be distinguished from those of neurons belonging to the other parts. Since it will be showed how it is possible to associate to each neuron an arbitrary *concept*, being able to distinguish among different neurons' behaviours could be somehow comparable to detect different concepts that are autonomously learned by the network, regardless to the original task it is trained on.

5.3 EXPERIMENTAL SETTINGS

This section describes the experimental setting adopted to study both the ability of the hidden neurons of a generic neural network in building an high-level representation for some specific source code-related features, similarly to what the authors of [79] and [114] did for images and natural language, and to characterize (and distinguish) the behaviour of the neurons in terms of their entropy and their activation patterns.

5.3.1 *Network Architecture*

The chosen work context is that of a simple neural model, namely an autoencoder. Without any hyper-parameters optimization nor any in-depth study on the network design, a simple dense autoencoder has been implemented, with two hidden layers in the encoder, two symmetrical hidden layers in the decoder and one code layer in the middle, as shown in Figure 12. The *ADADELTA* optimizer [151] and the Mean Squared Error as the loss function have been used. As the activation function for the hidden layers, the Rectified Linear Unit (ReLU) has been applied, which is defined, for all $x \in \mathbb{R}$ as $\max(0, x)$. The model is implemented using the APIs provided by the Keras library [29].

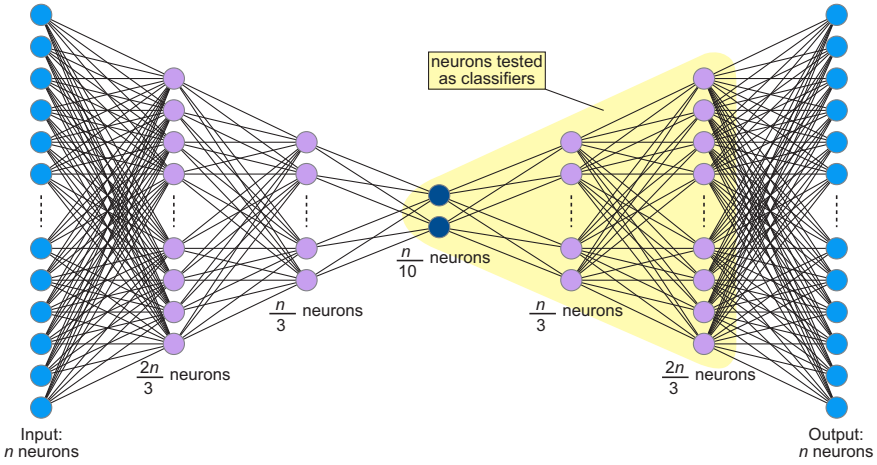


FIGURE 12: *Network architecture. The sizes of the layers depend to the program vectors dimension. In all these experiments,* $n = 300$.

5.3.2 *Dataset and Training*

For the training phase, the adopted dataset is that also used in [6], which is a collection of popular GitHub[1] Java projects that contains over 400000 methods, while for all the classification and ranking experiments the data are a subset of the Java-med dataset described in [8].

1 https://github.com

Two independent trainings of the autoencoder have been performed, by using two different source code embedding algorithms for preprocessing all the methods in the dataset and to compute the corresponding program vectors:

1. A 300-dimensional embedding obtained by simply applying the doc2vec model [78] to the methods in the dataset using the gensim framework [117]. To avoid inconsistencies related to formatting choices, such as the presence or absence of spaces between operands and operators, keywords and parentheses, the doc2vec model has been applyed to a pretty printed version of the methods obtained by using the Javaparser library [134].

2. The ast2vec source code embedding proposed in [126] and extensively described in Chapter 4.

Notice that this approach can be applied to any neural network, from the most simple to more sophisticated ones. In general, differently from the field of images, where the input of the network is exactly the object being studied, more often than with images, models for source code processing require a specific program representation as input, e.g. a numerical vector representation (embedding) or a stream of tokens, and each representation inherently preserves or emphasizes some program features to the detriment of others. Since in this discussion the main focus is to propose and evaluate the approach rather than the study of the best input representation, the two sets of program vectors considered in the experiments have different underlying construction ideas: doc2vec embedding is built by applying a classical NLP technique to the pure source code, so it is not supposed to be particularly viable in this context, while ast2vec is developed by considering both structural (neighborhood of nodes in the AST) and lexical (identifiers chosen by the programmer) features, and thus it is assumed to be particularly suitable and flexible for general program comprehension applications.

Each autoencoder has been trained for 50 epochs using the hyperparameters described in the previous section; due to the input vectors dimension, the layers in the encoder have, respectively, 300, 200 and 100 neurons, those in the decoder symmetrically have 100, 200 and 300 neurons, and the middle code layer has 30 neurons.

5.4 TASK-BASED EXPERIMENTS

For assessing the ability of the internal neurons in the considered networks to build internal representations for different program properties, each neuron has been tested when used as a classifier for distinct binary classification problems, following the same approach of previous works [79, 114].

5.4.1 *Problems Definition*

When dealing with images and product reviews (as in the referred papers), the properties according to which to classify the input objects can be easily defined: could be, for instance, the presence of particular patterns (e.g. cats or faces [79]) or positive and negative review sentiments [114], as in classical image recognition and sentiment analysis tasks. In the program comprehension context, however, such kind of properties do not directly arise from the source code, or at least they are not immediately evident for a human being reading it. Therefore, the first step was to define different labelling policies for classification, so to capture properties having different natures:

- the first one, designed using the Control Flow Graph (CFG) [7], addresses the syntactical structure of a method in terms of its *structural complexity*;
- the second one relies on the method's *identifiers* chosen by the programmers in order to target a task related to the functionality of a method;
- the third one is related to its *I/O relationship*, that is the relation between the input parameters and the returned object of a method;
- the last one is a *random* labelling strategy used as a baseline.

STRUCTURAL LABELLING POLICY The *cyclomatic complexity* [97] of a program can be defined starting from its CFG $\mathcal{G}$ having n vertices, e edges and p connected components as:

$$V(\mathcal{G}) = e - n + p \tag{1}$$

Dealing with java methods, such metric can be easily calculated by counting 1 point for the beginning of the method, 1 point for each conditional construct and for each case or default block in a switch-case statement, 1 point for each iterative structure and 1 point for each Boolean condition. Starting from this software metric, for a given parameter c, the problem can be defined as follows:

Problem 1 *Let M be a set of Java methods, let $c \in \mathbb{N}$, and let $h_c : M \to \{0, 1\}$ be a binary classification rule for the methods. We define, for each $m \in M$:*

$$h_c(m) = \begin{cases} 0 & \textit{if } V(\mathcal{G}_m) < c \\ 1 & \textit{otherwise} \end{cases} \tag{2}$$

IDENTIFIERS-BASED LABELLING POLICY For the definition of the semantic labelling strategy, the underlying assumption is the same that Allamanis et al. [6] and Alon et al. [9] made in their works,

namely that the name a programmer gives to a method can be somehow considered as a summary of the method's operations, meaning that the name of a method shall provide some semantic information on the method itself. Starting from this premise, the semantic labelling can be defined by considering the presence or absence of specific patterns, from a given set T, in the method name:

Problem 2 *Let M be a set of methods, let $N = \{lab_m : m \in M\}$ be the set of the names of the methods in M and let T be a set of patterns. We write $r \leqslant s$ if r and s are strings and r is a substring of s. Let $h_T : M \to \{0, 1\}$ be a binary classification rule for the methods. We define, for each $m \in M$:*

$$h_T(m) = \begin{cases} 1 & \text{if } \exists t \in T : t \leqslant lab_m \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

I/O-BASED LABELLING POLICY The idea beyond this kind of labelling is that the relation between the input and the output of a program can suggest something about the functionality of a program. For example, a program that takes an array of integers as input, and that returns another array of integers, could possibly be a program that fulfils some kind of sorting or filtering operations, while a program that requires as input an array, no matter its type, and that returns an object of the same type can possibly represent some kind of search operation.

For the definition of this class of binary problems, only a subset of all the possible I/O relations is considered, namely the presence or the absence of an array among the input arguments and whether the returned object is an array or a single element. For easing the discussion, the following binary notation to describe such possible relations is adopted:

00 : many to many

01 : many to one

10 : one to many

11 : one to one

Following this notation, this labelling strategy can be formalized as follows:

Problem 3 *Let M be a set of non-void methods, each having at least one input argument. Let $L = \{00, 01, 10, 11\}$ be the set of possible labels for each $m \in M$. We remark that it exists a function $l : M \to L$ that assigns a label to each method. Let $\mathscr{P}(L)$ be the power set of L. Let $h_P : M \to \{0, 1\}$ be a binary classification rule. We define, for each $m \in M$ and for a given $P \in \mathscr{P}(L)$:*

$$h_P(m) = \begin{cases} 1 & \text{if } l(m) \in P \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

RANDOM LABELLING POLICY Finally a baseline labelling strategy is defined to assess the obtained results, by comparing them with those obtained while solving an arbitrary task whose results should be only noise. Simply, consider a random split of the methods is considered:

Problem 4 *Let* L *be a randomly shuffled list of methods, and let* $m_i \in L$ *be the method having index* i*. Given a threshold* $n \in \mathbb{N}$ *and let* $h_n : L \to \{0, 1\}$ *be a binary classification function, we define, for each* $m_i \in L$*:*

$$h_n(m_i) = \begin{cases} 1 & \text{if } i \leqslant n \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

5.4.2 *Classification*

The performance of the hidden neurons has been tested in the classification of methods according to different instances of the classes of problems described in the previous section. To this aim, all the neurons in the code layer and in the two hidden layers in the decoder have been considered, as shown in Figure 12, and their classification accuracy have been tested by considering the activation produced by a neuron for each method given as input. The reason why only the decoding neurons have been tested lies in the nature of an autoencoder: in the encoder layers a progressive dimensionality reduction (and thus a compression of information) is performed, and this (likely) means that in the middle code layer only relevant features are encoded. Since in the decoder layers the dimension is symmetrically increased for reconstructing the input, only those neurons have been tested, since they are expected to hold more relevant features. Note that the same approach have been proposed, with promising results, for images [79] and for natural language [114], but it is new for source code processing applications.

In detail, for each of the selected neurons, the possible thresholds were 10 equally spaced values among the minimum and the maximum activation value of that neuron for methods in the training set. For each activation threshold, the classification accuracy of the neuron on a given problem instance has been computed by considering, in a precomputed balanced sample of the test set, the activation value of the neuron for that method and by predicting the method to be in class 0 or in class 1 if the activation value is less or greater than the threshold, respectively.

This process, formally described in Algorithm 1, supplies a procedure for ranking neurons according to a task: to each neuron is assigned its highest accuracy score. Table 7 shows the accuracies obtained by the best neuron for the considered problem instances, while the complete results obtained with the classification experiments will be discussed with further details in Section 5.6.

TABLE 7: *Best accuracy score for each problem instance*

Class	**Instance**	**doc2vec**	**ast2vec**
Random	none	54%	52%
Structural	$c = 10$	81%	84%
Semantic	$T = \{\text{test}\}$	63%	71%
Semantic	$T = \{\text{daemon}\}$	70%	68%
I/O	$\{00\}$ vs $\{01, 10, 11\}$	64%	65%
I/O	$\{00, 10\}$ vs $\{01, 11\}$	59%	64%

Algorithm 1 Algorithm for finding the best neuron in classifying programs on binary problems. The accuracy is computed by considering two balanced classes, each having at least 300 examples.

```
1:  bestAcc ← 0                                   ▷ accuracy of the best neuron
2:  for all neuron N do
3:      A ← activation values of N
4:      T ← activation thresholds                 ▷ 10 evenly spaced thresholds
    between 0 and max a ∈ A
5:      best_N ← 0                                ▷ best accuracy for N
6:      for all t ∈ T do
7:          pred ← empty list                     ▷ list of predictions
8:          for all a ∈ A do
9:              if a ⩽ t then
10:                 append 0 to pred
11:             else
12:                 append 1 to pred
13:             end if
14:         end for
15:         if ACCURACY(pred) ⩾ best_N then
16:             best_N ← ACCURACY(pred)           ▷ update best accuracy of N
17:         end if
18:     end for
19:     if best_N ⩾ bestAcc then
20:         bestAcc ← best_N                      ▷ update best neuron
21:         bestNeuron ← N
22:     end if
23: end for
```

5.5 NEURONS RANKING

The previous section presented the experiments for evaluating the ability of individual neurons in solving specific classification problems or, in other words, in recognising predetermined program properties. In the following, a scoring measure for neurons based on the concept of *entropy* used in information theory is presented. Further, the Mann-Whitney U statistical test [95] is used for comparing the behavior of two neurons, assessing whether they share similar activation patterns and thus whether they are able to approximately detect the same concept.

5.5.1 *Entropy of Single Neurons*

The method proposed for evaluating the importance of each neuron in the network is based on the information theoretical concept of entropy [131]. As it will be discussed in Section 5.6, the experiments performed for assessing the results obtained with this scoring approach proved the effectiveness of this ranking, since it can discriminate among neurons which exhibit very simple activation patterns (i.e. active on only very few instances, therefore with low entropy), from more elaborate ones (i.e. those having varied activation values on many instances, corresponding to medium or high entropy).

The baseline idea behind this approach is that each neuron can be seen as a signaling system whose symbols are its activation values. Formally, in information theory the entropy is defined as the average information obtained from a signaling system S which can output q different symbols $s_1, \ldots, s_q$ with probability $p_i = P(s_i)$:

$$H(S) = \sum_{i=0}^{i \leqslant q} p_i \log \frac{1}{p_i} \tag{6}$$

Dealing with activation values, whose domain is continuous over $\mathbb{R}_0^+$, a discretization of that space has been constructed by considering a set $R = \{r_1, \ldots, r_{1000}\}$ of 1000 evenly spaced intervals between 0 and the maximum activation value reached by a neuron for the vectors in the training set, and those intervals have been considered as the possible symbols of the neurons' alphabet. More precisely, for each neuron N the activation values yielded on a random sample of 10000 vectors have been computed, allowing to determine the number of occurrences of each symbol by counting the activation values yielded in each interval r_i. Then, the set $R_N \subseteq R$ of the occurring symbols in the neuron N has been considered, and for each $r_i \in R_N$ its occurring probability p_i has been derived by using a softmax function over the set of countings. Finally, to each neuron is assigned a score defined as the entropy computed over the set $P_N = \bigcup p_i$, where each p_i is the probability associated to the symbol r_i in R_N, as described by Algorithm 2.

Algorithm 2 Algorithm for computing the entropy of each neuron.

```
R ← list of intervals
for all neuron N do
    M ← random sample of 10000 methods
    V ← activations of N for each m ∈ M
    SCORE_NEURON(N, R, V)
end for

procedure SCORE_NEURON(N, R, V)
    C ← empty list
    for all r ∈ R do
        c ← number of v ∈ V such that v ∈ r
        append c to C
    end for
    remove all the 0s from C
    P ← SOFTMAX(C)
    return −∑_{p_i∈P} p_i log p_i
end procedure
```

5.5.2 *Pairwise comparison*

Further considerations can be provided on how entropy distinguishes neurons. The main insight is that it is always is possible to associate, to each neuron, a *concept* by simply looking at the instances that produces the highest activation values for that neuron. In other words, the concept corresponds to the binary classification task obtained by fixing an activation threshold and then by predicting the instances as satisfying that concept if the yielded activation value is higher than the threshold. Given this premise, it is possible to define an heuristic procedure for measuring the similarity of the concepts defined by two neurons, by applying the Mann-Whitney U test [95] in the following way:

1. choose two neurons N_{ref} and N_{cf}, representing the neuron that defines the concept and the neuron to compare it to, respectively,
2. considering the neuron N_{ref}, for each program instance $m_i \in M = \{m_1, \ldots, m_n\}$, compute the set of activation values $A = \{a_1, \ldots, a_n\}$ and create the list $L = \langle m_1, a_1 \rangle, \ldots, \langle m_n, a_n \rangle$, sorted according to a_i;
3. after splitting the sorted list L in three equally sized parts, generate the sets M_0 and M_1 by grouping the instances from the first and last of those parts, respectively;
4. select two equally sized random samples of instances from M_0 and M_1 and compute the corresponding two sets C_0 and C_1 of activation values for N_{cf};
5. perform the Mann-Whitney U test on the sets of values C_0 and C_1, with alternative hypothesis that the distribution underlying

the first set is stochastically *less* than the distribution underlying the second one.

Notice that, in the outlined procedure, the threshold is represented by a range of values instead of a single point. The reason is to make the definition of the binary classification problem more robust, since the points in the middle that could likely give rise to confusion are removed.

As it will be discussed in Section 5.6, the application of this procedure to all the possible pairs of neurons allows to assess that different entropy values correspond to different behaviours in terms of recognised concepts. Further, the replicability of the concepts defined by the neurons varies when comparing neurons belonging to different entropy ranges.

5.6 RESULTS

This section introduces the results obtained with outlined experiments. A first analysis concerns the performance of the neurons while solving different instances of the classification tasks described in Section 5.4, the second part is a study of the neurons from the information theoretic standpoint discussed in Section 5.5, while the third part reports the comparison of pairs of neurons and shows how different entropy intervals clearly characterize specific behaviours.

5.6.1 *Classification Experiments*

A summary of the results obtained in the classification experiments is reported in Table 7. Different instances for the classification problems described in Section 5.4 have been considered, and the classification accuracy of each neuron has been evaluated. As can be seen in Figure 13, where the accuracy distributions for some of the considered problem instances are reported, for each problem most of the neurons reach an accuracy level between 0.5 and 0.55, while only few neurons are indeed able to reach higher accuracies. This evidence is already interesting by itself, since it means that single neurons perform differently when tested on a given task and also that some neurons are actually able to detect source code related properties. The accuracy varies a lot when considering different problem and different embeddings, but this is probably due to the features that are naturally seized by the vectors. Indeed, in the experiments this is confirmed by the good results obtained for the structural task with the ast2vec embedding (first diagram in Figure 13). Finally, the experiments on the baseline random problem confirm the validity of the results by showing how the performance of the neurons on a randomly defined problem is far from being comparable to the one obtained on all the other tasks, hence good performances are not emerging by chance.

5.6.2 *Entropy-based Experiments*

The original goal of entropy in information theory is to measure the average level of information of a source, given its outcome. In this setting, each neuron can be considered a source of symbols, when its possible activation values are interpreted as explained in Section 5.5.

The defined entropy allows to distinguish (internal) neurons with respect to the variety of the activation values they output for each instance presented to the network. Eventually, each neuron's behavior in terms of output activation values is defined by the training process and, when considering internal neurons, also by its connectivity to the rest of the network.

Here the perform experiments are aimed at showing how such a measure can be used to identify neurons that can be used to perform some interesting classifications of input instances. Previously, interesting neurons have been identified by first specifying some classification problem, and then by measuring the performance of each neuron on it; differently, the interest here is to specify what can characterise interesting classifications, to look for corresponding behaviors among the internal neurons.

In this case, the goal is to understand how the behavior of neurons varies with different entropy values, and when operating on two different ways of embedding source code input instances.

To this aim, the distribution of entropy values among the neurons of the autoencoder's section highlighted in Figure 12 has been plotted. The resulting distributions, in Figure 14, are qualitatively similar under both doc2vec and ast2vec embeddings, with a bimodal profile characterized by a peak of occurrences for very low values of entropy and an area of normally distributed frequencies for higher entropy values.

Therefore, three classes of neurons having different entropy values can be roughly distinguished:

1. A big number of neurons (notice that the figure is in a logarithmic scale) having an entropy equals or very close to 0. These neurons are of no interest in this context, since they are neurons that (almost) never activate. They could only be used for pruning the network in order to optimise the architecture, but it is out of the focus of this work.

2. Another big class of neurons having normally distributed high entropy values. Those neurons reach an high score since their activation values are distributed over a wide range. In addition, the probabilities of the occurring activation values to be in distinct intervals are relatively similar: this leads to an high score in terms of information theory.

3. A smaller set of neurons whose values are higher than 0 but that are out of the normal distribution of the majority of the values. The corresponding activation values are those between

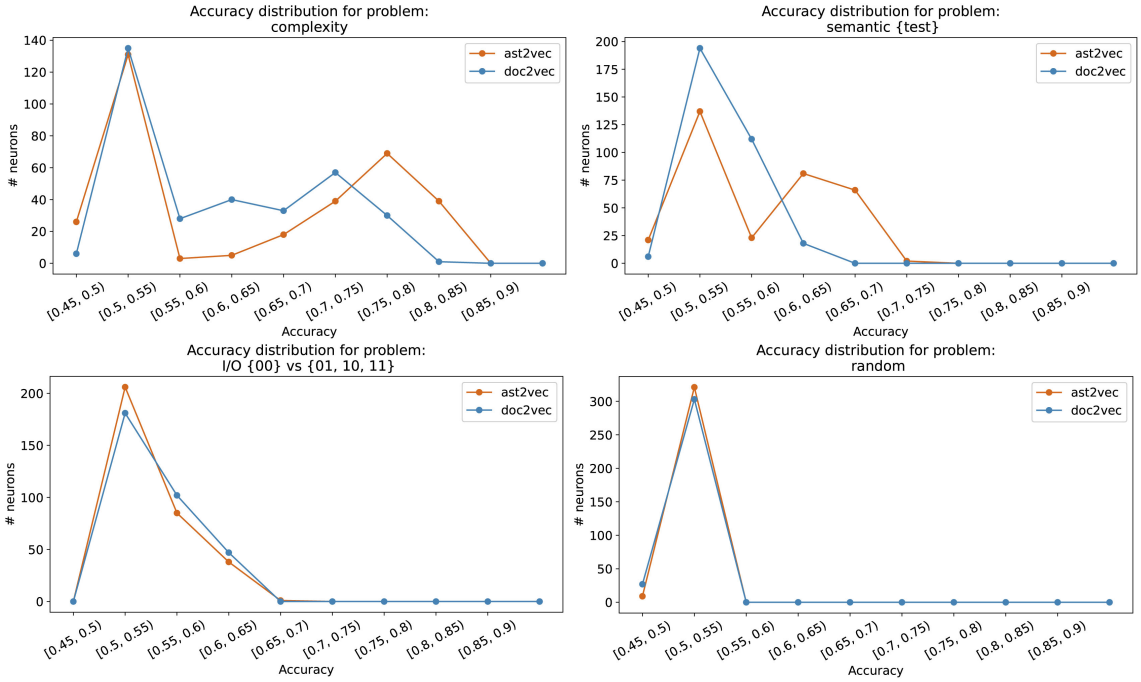


FIGURE 13: *Classification accuracies reached by the neurons on different problem instances.*

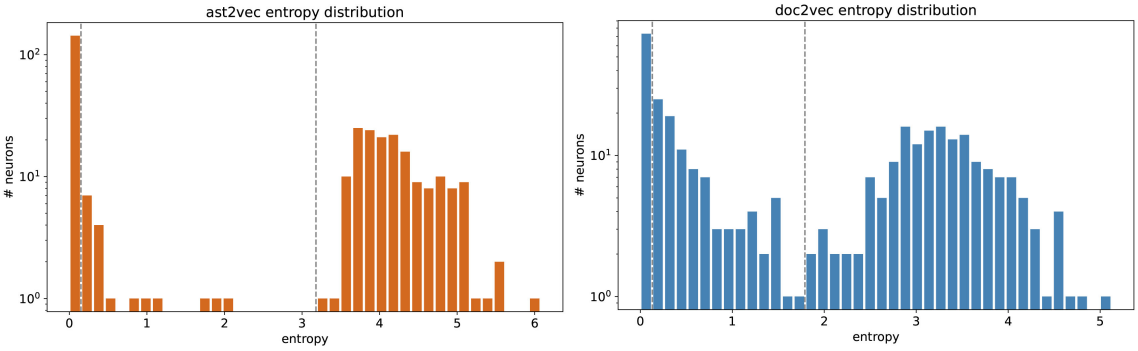


FIGURE 14: *Distribution of the neurons' entropies in the ast2vec (left) and doc2vec (right) models.*

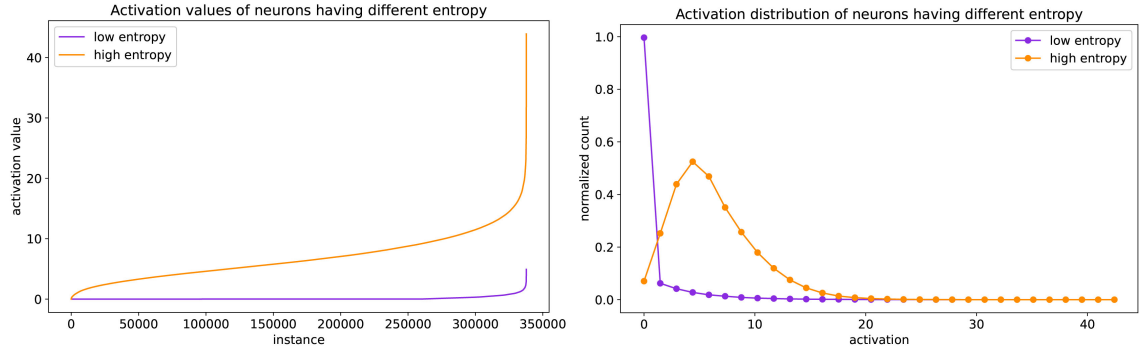


FIGURE 15: *Activations of neurons having different entropies. In both the images, the purple line refers to a neuron having a low entropy score* $H = 1.06$*, while the orange line represents a neuron with an high entropy* $H = 5.19$*. The left figure is a simple plot of the sorted activation values for the two neurons, while the left figure reports the distribution of the activation values.*

the dashed bars plotted in Figure 14. Those neurons are peculiar, since they produce an activation higher than 0 only for a significant amount of vectors (i.e. $> 10\%$), while in all the other cases their activation is equal to 0.

For two neurons, one with low entropy and one with high entropy, we plotted their activation values for a set of input instances in Figure 15 (left). More related to our entropy measure, we show in the same Figure 15 (right) the distribution of symbols, defined on intervals of activation values, for the same two neurons.

5.6.3 *Pairwise Comparison Experiments*

To explore what neurons are representing, the investigation has been performed through the comparing measure introduced in Section 5.5.2. The insight was to take two neurons, N_{ref} and N_{cf}, to assign to N_{ref} a reference role, to compare the distribution of the activations of N_{cf} with that of N_{ref} over the same set of instances, and to discuss about how they classify input instances. Preliminary findings show that experiments with ast2vec and doc2vec produce very similar results, for this reason all the following discussion is made by considering only the neurons in the ast2vec autoencoder.

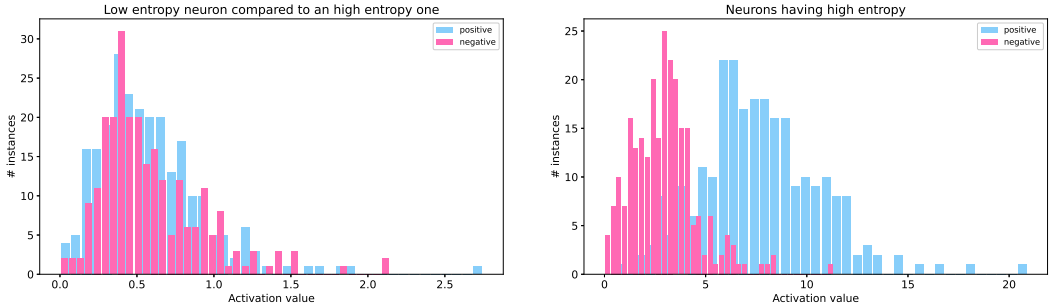


FIGURE 16: *Activations of two high-entropy neurons, one compared against a low-entropy neuron (left), and one against a high-entropy neuron (right). Colors of bars correspond to the classes M_0 and M_1 defined by the reference neuron N_{ref}.*

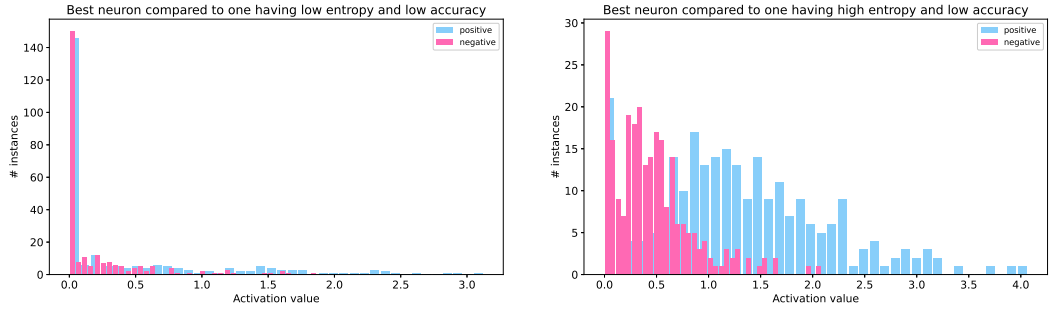


FIGURE 17: *For the cyclomatic complexity classification task, outcome of the comparison between the best neuron and two neurons performing bad on the same task, one having low entropy (left), and the other having high entropy (right).*

As stated in Section 5.5.2, the activations of a neuron over a set of instances define its classifying behavior, thus two neurons can be compared in terms of how a neuron can approximate the classification of another. For instance, looking at Figure 16 (right), we can see how a chosen neuron N_{cf} activates on negative instances (set M_0, bars in red) or positive (set M_1, bars in blue) with respect to the classification yielded by N_{ref}. In this case, it appears that it is possible to find a threshold on activation values that is good to classify most of the instances in the same way as the chosen N_{ref}. Notice that, in general, such a threshold could be found when the medians of the two distributions, one from the activation values of N_{cf} on M_0 and the other from its activations on M_1, are separated.

This fact can be assessed by operating the Mann-Whitney U test on those pairs of distributions. Remark that the outlined results still stay valid when considering the alternative hypothesis of having the first distribution stochastically *greater* than the second one. In fact, this would mean that instances producing a high activation level on the first neuron tend to produce a low activation level on the second one, and viceversa. Therefore, in this case the binary classification produced by the first neuron could be approximated by the opposite of the classification produced by the second one.

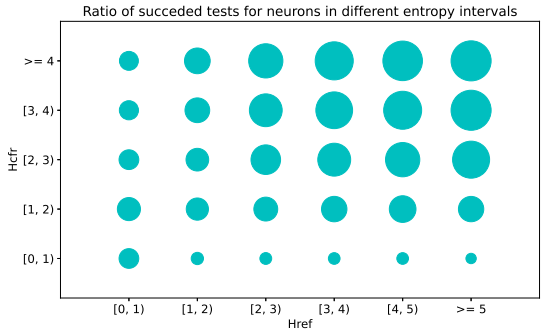


FIGURE 18: *Ratio of successful Mann-Whitney U tests while comparing neurons belonging to different entropy ranges.*

Overall, the reproducibility of the classification yielded by a neuron N_{ref} by a second neuron N_{cf}, has been explored on every pair of neurons from the autoencoder section previously considered. A first evidence is that there is a correlation between the entropies of the two neurons compared as described above. In Figure 18 the success ratio of the Mann-Whitney U test according to the entropy of the neurons being compared is pictured. When both the reference and the comparison neurons have high entropy, Mann-Whitney U test favours the *less* hypothesis, with $p < \alpha$ where $\alpha = 0.01$ is the considered significance level.

For instance, Figure 16 shows the details of two comparisons, one (left) where a neuron with low entropy is compared to a neuron with high entropy, and another (right) where two neurons having high entropy are compared. The bars represent the distribution of the activation values for the instances in the two classes M_0 and M_1 defined

by N_{ref}. Specifically, the colors of the bars are associated to the classification of instances of N_{ref}, while their position and height represent the activation levels produced by N_{cf} on the same instances. The two distributions on which the Mann-Whitney U test is performed are the activation values of the N_{cf} for instances associated to the two sets M_0 and M_1, respectively. In the first plot (left) the comparison of two neurons over which the test fails is reported, meaning that it is not possible to distinguish among the two distributions and thus it is probable that the two medians are not one less than the other. This can be read as an evidence that no threshold is good enough to separate red and blue bars, while in the other one (right) this can be done with some approximation.

As a final check, we verified how those measures apply on neurons which performed well in one of the specific tasks presented in Section 5.4.1. When considering the neurons having accuracy above 0.8 on the task of estimating the cyclomatic complexity of source code, the gathered evidence shows that:

- over the total 330 neurons, 39 under ast2vec embedding reach such accuracy, but only 1 for doc2vec;
- the neurons that reach this high accuracy 3 all have an entropy hicher than 3;
- the converse is not, in general, true: some of the neurons having high entropy perform badly on the same task;
- chosen a neuron performing well on the task, its comparison with low entropy neurons always fails (see Figure 17 left), and the outcome of its comparison to medium or high entropy neurons can be related to how good is their accuracy on the task (see Figure 17 right).

With the proposed entropy measure and the comparison based on Mann-Whitney U test, it is possible to first select and then group neurons which could be considered to be representative of specific learnt concepts. In general, neurons having high entropy exhibit behaviors that appear to be the most interesting, and they can be compared, with respect to the concepts they can recognise, to neurons toward which the test succeeds.

5.7 FINAL REMARKS

The work described in this chapter can be considered as a seminal investigation of the dynamics that rule the internal parts of a neural network, with the aim of extracting knowledge from there. The problem has been approached across the specific application domain of source code analysis, where what can be considered as "interesting" can not be defined as easily as when, for instance, recognising objects in the natural domain of images.

The approach is twofold: while in a first part the performance of neurons – that belong to a network that has been independently

trained – is measured when using them as classifiers for tasks related to properties of source code snippets, successively, in the same network, the internal dynamics are studied according to information theoretic and statistical measures.

5.7.1 *Insights and Limitations*

The investigation has been performed by using a simple autoencoder as reference model, and its input was obtained from source code by using two different embeddings, one that just considers the linear sequences of words in the code (i.e. as if dealing with natural language), and another that takes into account also the formal parsing of the programming language. The findings tell that:

- several internal neurons perform well on tasks related to syntactic properties of code, such as that of cyclomatic complexity;
- the two embeddings perform differently, when considering different tasks;
- all the neurons that perform well on known tasks, also reach high entropy values;
- by choosing appropriate thresholds on activation values, neurons with high entropy are able to approximate each other's behavior.

Despite the promising results, this work as it stands presents some limitations. First of all the chosen neural network, namely the autoencoder described in Section 5.3.1, should be too simple to capture all the source code properties needed for a deepened analysis. Moreover, the autoencoder operates on a "transformed" version of the source code (namely the program vectors), and thus the results obtained, which result from the information given by the data after the embedding process, are affected not only by the neural model under analysis but also by the chosen neural embedder. For the purposes of this work, which is focused more on the method than on the results, these are not major issues, but they need to be taken into account in the future research on this topic.

5.7.2 *Further Directions*

The most spontaneous sequel for this work consists in applying the outlined methods to explore the behavior of internal neurons of more sophisticated networks since, for instance, as already discussed in Chapters 2 and 3, recent works showed how neural transformer models can be fruitfully used on source code [2, 65]. Moreover, these kind of architectures do not need particular source code preprocessing, and thus both the major limitations would be overcome. The introduced techniques could be employed to look for neurons which perform well on known tasks, even if belonging to a network trained

while keeping in mind other goals. Thereto, it would also be interesting to reason on group of neurons instead of single ones, for instance by analysing how they classify with respect to the richness of their activation patterns, and to group them by similarity, as allowed by the information theoretical measures.

Furthermore, the insight of exploiting the knowledge yielded by the neural activations has been also leveraged in other works that comprise this thesis, as it will be evident in the Chapters 6 and 7. Finally, as it will be unfolded in Chapter 8, it is possible to move at an higher abstraction level, and study how to associate human understandable concepts to the discovered internal activation patterns, similarly to what other authors did for instance in the fields of image understanding [71] and of chess [98]. The natural expectation is that, the more the neural model is structured and powerful, the more such kind of measures can prove their effectiveness in studying the internal representations and the developed knowledge of neural systems. This is an intriguing issue that deserves further research.

6

EVOLUTIONARY APPROACHES FOR ADVERSARIAL ATTACKS

This chapter, that is based on the work published in [50] presents an evolutionary approach for assessing the robustness of a system trained in the detection of software vulnerabilities.

The method is based on the application of a Grammatical Evolution genetic algorithm, and on the use of the output of the system being assessed as the fitness function. Through this combination it is possible to easily change the classification decision (i.e. vulnerable or not vulnerable) for a given instance by simply injecting evolved features that in no wise affect the functionality of the program. Additionally, by means of the same semantic-preserving modification technique, one can significantly decrease the accuracy measure of the whole system on the dataset used for the test phase.

6.1 ROBUSTNESS AND ADVERSARIAL ATTACKS

In the literature, as already discussed in Chapter 2, there exists a wide range of machine learning systems used for solving different source code-related tasks. The promising results obtained with such models suggest that ML systems (and especially neural networks) are able to learn valuable and diverse properties related to the source code. This chapter presents a study whose founding idea is to exploit the knowledge learned by ML systems dealing with source code for leading the evolution of programs that satisfy given specifications or bear certain properties.

The carried out investigation arises from an adversarial perspective, in which such knowledge is used to lead the generation of that could deceive a vulnerability detection system. In general, statistical classification, namely the problem of assigning, given a set of instances and a set of categories, an instance to the correct category, has been widely studied in the years, especially with the diffusion of machine learning methods since the '80s [77].

The interest is also in the field of cybersecurity: as simple examples, phishing and spam detectors [66] but also malware detectors [63] are all evidences of the importance of being able to effectively distinguish between what is safe to what it is not, and thus to classify objects such as emails, files and programs. However, binary classifiers are often susceptible to classification errors. While in some cases such misclassifications only affect the accuracy measure, in some scopes the presence of false positive or false negative is crucial; just think, as an example, to antivirus: in this context, a false positive malware alert it is not a big deal, while a false negative could be critical. For this reason, evaluating the robustness of a classifier is essential. If a classifier system is widespread, it is possible that an attacker looks for

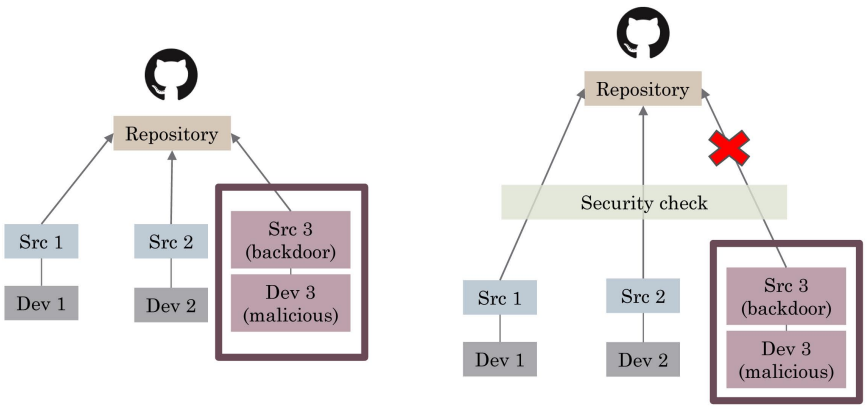


FIGURE 19: *Possible attack scenario.*

features of input source code that make it evade the surveillance. The system has to be evaluated with respect to such risks. When this task is related to image classification, where instances are vectors of values from a continuous domain (pixel colors), the approach is to check the results of small variations of input values. The challenge is however harder when instances are source code fragments, where each variation on their feature values is actually a variation of a syntactic (sub)tree, which must obey the grammar of the chosen programming language.

This study proposes an evolutionary approach for assessing the robustness of a classification system by producing program instances that mislead a system trained in the detection of software vulnerabilities. In other words, the goal is to synthesise vulnerable programs that will be classified as safe by the system, or vice versa. A possible application scenario is shown in Figure 19, in which many developers make commits in a shared repository, and one of them is malicious and wants to insert a backdoor. In an unsafe setting (left) the backdoor can be easily placed. In the setting reported in the right image, instead, a security check is installed, and the malicious code should be blocked. The goal of the attacker is thus to deceive the check and pushing malicious code that is not detected. The work resulted in the following contributions:

- A novel evolutionary approach for synthesising programs using, as the fitness function, the output of a neuron in a neural model.
- The application of this approach in the construction, or adaptation, of a dataset so that it leads to a high percentage of false negative (or, possibly, false positive) misclassifications on a network trained in the detection of software vulnerabilities.
- The use of the best individuals for characterising the syntactical features that mainly stimulate a given neuron.
- How such characterization can be used for modifying the input instances of a classifier in order to arbitrarily change the classification decision.

The source code to replicate the experiments described in this chapter is available in a public repository[1].

6.2 WORK OVERVIEW

The founding idea of this work is to synthesise input instances which are able to lead the network to a wrong classification decision, by exploring the space of possible solutions by means of a GE algorithm, using the numerical output of the model as the fitness function. This insight allows to efficiently explore the behaviour of a classifier by leveraging an evolutionary pressure instead of randomly sampling the solution space, and thus to possibly discover some blind spots or some features that can mislead the model.

The authors of [122] propose a tool for detecting software vulnerabilities using a deep feature representation based on a lexical tokenization of the source code. In that work, an approach derived from classical NLP techniques for sentiment analysis [72] is developed for classifying programs which are potentially vulnerable to known categories of software weaknesses (CWE[2]). Using this model as a benchmark, three classes of experiments have been designed:

- The evolution of *pure* individuals able to maximize or minimize the output of the network, as summarized in Figure 20. In these experiments, the fitness function is computed as the output of the network given the evolved individual as the input.
- The injection of evolved individuals in functions of the original dataset, in a way that does not affect the behaviour of the function (i.e. after the return statement, as will be explained in Section 6.3). In this case, the fitness function is computed as the output of the network given as input the *hybrid* program consisting of the original functions modified with the injection, as outlined in Figure 21. The aim of these experiments is to arbitrarily change the classification decision for a given instance, namely to maximise the fitness if starting from functions classified as negative and vice versa.
- The injection of evolved individuals in all the original test set, in order to change the classification statistics of the network (Figure 22). The goal is to assess the approach by analysing how accuracy, precision and recall change depending on how and which evolved individuals are used for the injection.

For ease of reference, below is reported some terminology that will be used in the rest the paper:

INDIVIDUAL An evolved program $\mathcal{P}$. It consists of a snippet of code that complies with the formal grammar specified in Figure 25. Examples can be found in Figures 24 and 23.

1 https://github.com/Martisal/adversarialGE
2 https://cwe.mitre.org/

PURE INDIVIDUAL An evolved program $\mathcal{P}$ whose fitness function $F : \mathcal{P} \mapsto [0, 1]$ is computed as the output of a neural model given $\mathcal{P}$ as input.

HYBRID INDIVIDUAL An evolved program $\mathcal{P}$ whose fitness function $F : \mathcal{P} \mapsto [0, 1]$ is computed as the output of a neural model given as input an instance $\mathcal{P}_0$ from the dataset after the semantic-preserving injection of $\mathcal{P}$.

SEMANTIC-PRESERVING INJECTION The addition of instructions in a program that do not alter the *semantic* of the program itself. In this work, only the following transformation is considered: given two programs $\mathcal{P}_0$ and $\mathcal{P}$, we call the semantic-preserving injection of $\mathcal{P}$ in $\mathcal{P}_0$ the addition of the instructions contained in $\mathcal{P}$ after the return statement of $\mathcal{P}_0$.

POSITIVE (OR NEGATIVE) INSTANCE This work always refers to the vulnerable (or safe) programs in the dataset as positive (or negative) instances. It will be clear in turn if positive (or negative) refers to the ground truth or to the classification decision of the model.

6.3 EXPERIMENTS

The experiments are organized along two directions: the use of GE to evolve source code fragments, forcing their binary classification by a given trained neural network, and the check of how selected evolved individuals could be good to alter the classification of any instance from a labeled dataset. The goal is to show how easily an attacker can mask the vulnerabilities in any source code instance, and thus to foil the classification by a neural network, even if treated as a black box.

This section reports the technical details of the performed investigations, from the existing models and libraries used, to the design and execution of the experiments.

6.3.1 *Vulnerability Detection Model*

The deep learning classifier presented in [122] has been considered as a benchmark, since it is quite renowned in the literature, and it exhibits a good performance (94% accuracy) after having replicated the training on the dataset curated by the same research group[3]. It should be finally remarked that all the experiments have been performed on a test partition of the same dataset, which has not been used during the training phase.

The considered neural classifier is based on a preprocessing of source code where each instance, i.e. a C language function, is tokenized by a lexer, and then fed into a deep stack of feedforward layers, including a 1-dimensional convolution layer, which terminates in

3 Available at https://osf.io/d45bw/

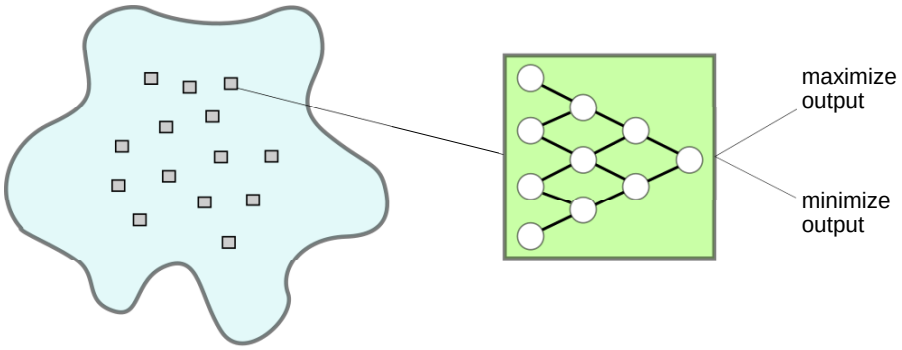


FIGURE 20: *First class of experiments. Fitness function is the output of the network given the pure evolved individual as input.*

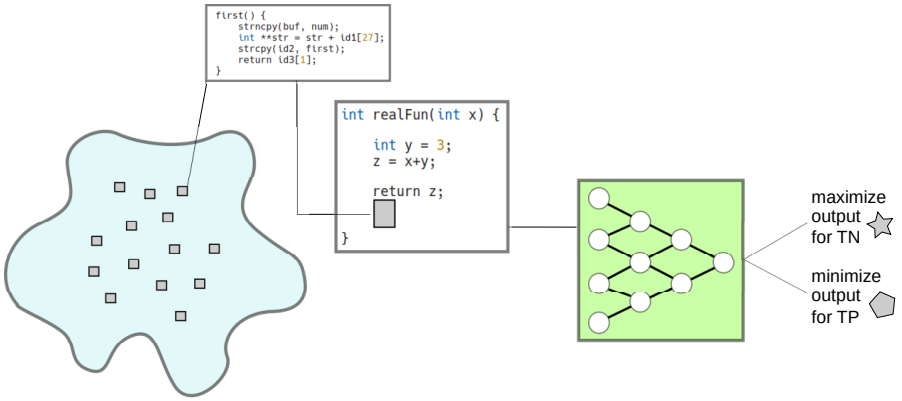


FIGURE 21: *Second class of experiments. Fitness function is the output of the network given an hybrid input obtained by injecting an evolved individual in a true positive (if minimising the fitness) or a true negative (if maximising the fitness) instance of the original dataset.*

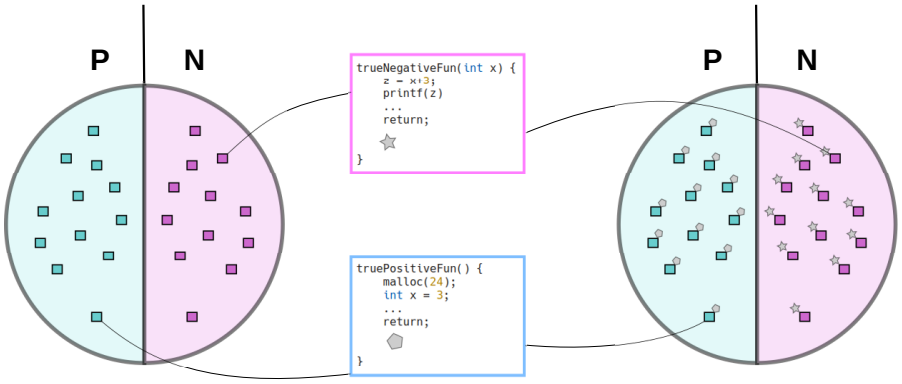


FIGURE 22: *Classification experiments. The original dataset (on the left) is modified to generate a new dataset (on the right) by injecting, in all the instances, an evolved individual.*

```
int id4() {
    do {
        id2 -> id1 = id1 + argv[52] / 6;
        if (id2[7] + b <= argv[29] / id3[9] / buf[9] - argv[54]) {
            buf -> first = 8 / first[54] / -883 + id1;
            if (5 - argv[8] / id3 - argv[66] != id1 / -1621 / id1) {
                if (id2[7] + b <= argv[29] / id3[4] / -896 + buf -> id2) {
                    char *first = e / -893 / id1 / -5;
                }
                **id1 = first[028] / first[54] / -883 + id1;
            } else {
                if (first[4] + num[893256] == -5 + e / first[028] / first[54]) {
                    int **id3 = num -> str; num = argv[40] * buf;
                } id3 = num[548]; int id2 = id1[6] * id1 / -1631;
                id1 = 147 + b; num = buf[293] - argv[8] / buf[9] - argv[40];
            }
            puts(buf);
        }
    } while (num -> id2 == -6 * e / 147);
    return 16;
}
```

FIGURE 23: *Example of an evolved individual, minimised fitness function.*

```
int main(int argc, char **argv) {

    char num[0];
    return 347;

}
```

FIGURE 24: *Example of an evolved individual, maximised fitness function.*

```
<function-definition> ::= <type-specifier> <fdeclarator> { <statements> return <operation>;}
                        | void <fdeclarator> { <statements> return;}
                        | int main(int argc, char **argv) {<statements> return <operation>;}

<operation> ::= <primary-expression>
              | <primary-expression> <operator> <primary-expression>
              | <primary-expression> <operator> <primary-expression> <operator> <primary-expression>

<statement> ::= <type-specifier> <declarator> = <operation>;
              | <type-specifier> <declarator>[<digits>];
              | <declarator> = <operation>;
              | <identifier> -> <identifier> = <operation>;
              | <selection-statement>
              | <iteration-statement>
              | <custom-statement>

<selection-statement> ::= if (<boolean-expression>) {<statements>}
                        | if (<boolean-expression>) {<statements>} else {<statements>}

<parameter-list> ::= <type-specifier> <declarator>
                   | <parameter-list>, <type-specifier> <declarator>

<iteration-statement> ::= while (<boolean-expression>) {<statements>}
                        | do {<statements>} while (<boolean-expression>);

<fdeclarator> ::= <identifier>(<parameter-list>) | <identifier>()

<identifier> ::= str | buf | first | num | id1 | id2 | id3 | id4

<declarator> ::= <identifier> | <pointers><identifier>

<constant> ::= <integer-constant> | <character-constant>

<statements> ::= <statement> | <statements> <statement>

<boolean-expression> ::= <operation> >= <operation>
                       | <operation> <= <operation>
                       | <operation> == <operation>
                       | <operation> != <operation>

<primary-expression> ::= <identifier>
                       | <constant>
                       | <identifier>[<digits>]
                       | <identifier> -> <identifier>
                       | argv[<digits>]

<custom-statement> ::= gets(<identifier>);
                     | puts(<identifier>);
                     | strcpy(<identifier>, <identifier>);
                     | strncpy(<identifier>, <identifier>, <digits>);

<character-constant> ::= 'a'|'b'|'c'|'d'|'e'

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<integer-constant> ::= <digits> | -<digits>

<digits> ::= <digit> | <digits><digit>

<type-specifier> ::= char | int

<pointers> ::= * | **

<operator> ::= +|-|*|/
```

FIGURE 25: *BNF C grammar used in the experiments.*

a single output value ranging from 0 to 1. The classification of an instance is positive (vulnerable) when the output value is higher than 0.5, negative otherwise. In particular, the network has been trained on the supervised task of recognising vulnerabilities of type CWE-120 (classic buffer overflow).

The dataset, that consists in over one million labeled C functions, has been split in three folds, with ratio 80-10-10, used respectively for training, validation and testing. Notice that each of these folds has a number of instances strongly unbalanced between the two classes. During the training, the number of instances of the two classes has been kept in a ratio of 5 negative instances for each positive instance.

As previously mentioned, the trained model exhibits a good accuracy of 94% on the test set, though among the available performance indices the precision/recall area under curve (P/R-AUC) [36] has been considered, since it is more suitable for dealing with unbalanced sets of instances. The trained network has a P/R-AUC index of 0.42 on the chosen test set.

All the experiments have been done on a Linux machine with 16GB RAM and a Nvidia GTX 1070 GPU. The training phase took around 3 hours.

6.3.2 *Grammatical Evolution*

The library used for evolving the individuals (consisting of C language functions) with GE is PonyGE2 [49]. The grammar, which is reported in Figure 25, has been derived from a realistic C language grammar by reducing the set of productions mainly to: (i) declaration and use of integer and char variables, (ii) use of pointers and arrays, (iii) single function declarations and (iv) call of functions from a small set of standard C functions. All the evolutionary runs have been performed using generational replacement with elite of size 1, tournament selection (tournament with size 3), with mutation and one-point crossover probabilities of 0.1 and 1, respectively. Finally, the fitness of individuals is computed by preprocessing each of them as required by the neural classifier (i.e. by using the same tokenizing lexer built during training), and then by reading the real valued classifying output of the network.Therefore, the fitness function that guides the evolution of individuals is a value between 0 an 1, where values above 0.5 meant that the individual is classified as vulnerable. All the runs evolved for 33 generations, and individuals had a maximal tree depth (referring to the expansion of non-terminal symbols) limited to 30.

It should be noted that the tokenization considered only the 10000 most frequent words from the dataset, including C identifiers, operators, keywords and punctuation. Therefore, the grammar used to generate GE individuals (Figure 25) was designed to produce both known (tokenized) identifiers as well as new ones, which are ignored by the tokenizer. For instance, the non-terminal `<identifier>` in the grammar includes both `buf`, which appears in the vocabulary of the

tokenizer defined during training, and `id1` which does not. This design choice is made in order to allow the GE to sample the input space with more freedom, so that adversarial examples could go beyond what the training set offered as vocabulary. Each evolutionary run was processed on the same machine used for the training phase, and took less than 1 minute.

6.3.3 *Evolutionary Runs*

Given the two experimental platforms described, namely a GE system to evolve C language functions, and a deep network trained to recognize buffer overflow vulnerabilities, a set of experiments has been performed to assess the weaknesses of the neural classifier.

PURE INDIVIDUALS The first investigation was to verify how the GE system performed when asked to define C functions classified as vulnerable (or not vulnerable), that is to maximise the fitness function (or to minimise, respectively), with its value defined by inferring the output yielded by the classifier given as input an evolved individual from the population. Even with the simplified grammar described in the previous section, both the goals (neural output value above or below 0.5) have been easily reached on each evolutionary run. As an interesting finding, it can be noticed that C functions maximising the fitness have been consistently short (less than 100 characters), while the individuals minimising it were always much longer (see examples in Figures 24 and 23).

The individuals produced in this first experimental part will be referred to as *pure*, since they are evolved with no reference to C functions from the dataset, and will be considered as *positive* when maximising the fitness, and as *negative* in the opposite case.

HYBRID INDIVIDUALS The second set of experiments has been aimed at evolving, by GE, C source code snippets to be injected in existing instances. First, from the dataset some instances labeled as vulnerable and some labeled as not vulnerable have been selected. Then, starting from a single instance taken from the dataset, GE has been applied to evolve C snippets to be inserted in the starting instance, whose fitness is computed by inferring the output of the model given as input the original instance, modified with the evolved individuals. In other words, the fitness of an individual is measured after the injection of that individual in the original instance (see Figure 21). Notice that the individual was inserted in a way that the actual behavior of the given function is not affected: the evolved snippet is appended inside the instance right before the closing bracket, after any return, effectively becoming *dead code* with no effect at compile or run time. Other simple semantics-preserving transformations could also be used, such as blocks controlled by a condition which will never be true. The same has been done both for positive and for negative in-

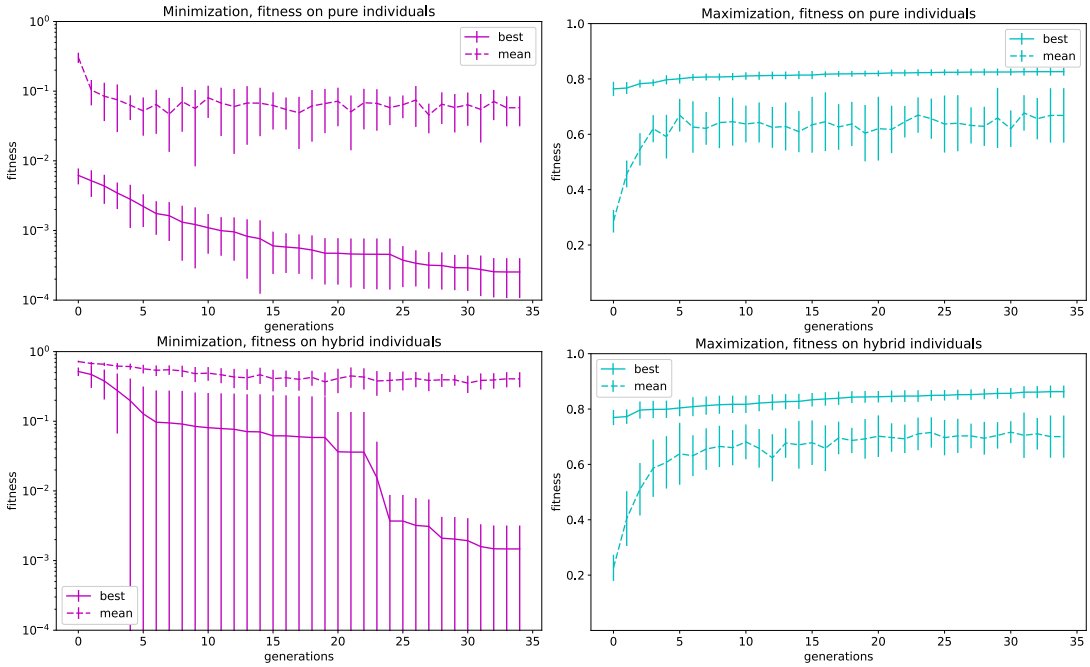


FIGURE 26: *Fitness of the evolved individuals through the generations. Plots on top refer to pure individuals while those on bottom to hybrid individuals. Plots are built by considering the mean of the fitness values obtained over 10 runs, with vertical bars reporting the standard deviation. Dashed lines indicate the average fitness of the population, while continuous lines indicate the fitness of the best individual. Notice that, for the minimization experiments, plots are in logarithmic scale.*

stances, and in the following such modified instances will be referred to as *hybrid* individuals.

In this way, the possibility of masking a vulnerability in a function to be classified is investigated, or conversely to force a positive classification of a function that is actually not vulnerable. Every experiment in this set has been successful, always finding an individual which, when injected in a given labeled function, made that same function be wrongly classified by the neural network. It can be observed that in this setting, the winning individuals were more complex than those evolved as pure. The progress towards individuals with the wanted fitness (positive or negative) was still quite fast during each GE run, as it can be seen in the plots in Figure 26.

6.3.4 *Classification of Evolved Instances*

Finally, the set of C source code fragments produced by GE, has been used to assess the robustness of the trained neural network in recognizing vulnerabilities in known instances. The first check has been done by injecting pure individuals in known instances, with these obvious cautions: from the evolved functions, the body has been extracted and inserted in the instances after the return statement, to obtain a function which is different, but syntactically correct and with the same original behavior (explanation in Section 6.2).

The chosen pure individual was always generated with a target fitness corresponding to a classification which was opposite to the true

classification of the given instance. The results, reported in Tables 8 (row PureMaxMin) and 11c, showed that it is possible to deceive the network with high probability, by leading it to classify the modified instances in a way opposite to the true label. The P/R-AUC index on the set made of injected instances indeed drops from 0.42 to 0.06, as it can be seen by comparing it with that reported in row Original of Table 8 and with Table 9.

Some performance difference between the evolutionary building of pure individuals, where the fitness was evaluated just on the pure individuals, and the classification of the test set made of transformed instances, where each modified function was evaluated, was expected. With the experiments conducted on hybrid individuals, however, the performance of the neural network has been assessed on a test set where each instance was modified with a code fragment evolved with GE when the objective was to change the classification of a single original given instance. Such modifications, although made by means of individuals evolved to perform in the context of a different given function, has been successful since the neural classifier showed a worsened performance measured by a P/R-AUC down to 0.02, as reported in Tables 8 (row HybrMaxMin) and 11d. As a further evidence of the statistical significance of our experiments, the last column of the table reports the p-values obtained with the Mann-Whitney U test [95] on the different, modified, datasets compared with the original one. In all cases, assuming a significance level $\alpha = 0.01$, the null hypothesis can be rejected, stating that the proposed evolutionary approach is always able to produce solid adversarial examples. A complete overview of results can be found in Tables 8 and 10, where more details are reported, in terms of both performance indexes and confusion matrices. Confusion matrices, specifically, highlight how each injection experiment changes the classification decision for the test instances, towards being wrongly classified as positive or negative, according to the goal of each specific experiment.

6.4 DISCUSSION

This final section discusses the experimental results to show how the proposed technique is able to build an effective attack to the classification task of a given machine learning system.

6.4.1 *Threat Model*

To assess the relevance of our results, we first describe the threat model [109] (or the assessment goals) we have considered, with respect to a given classifying machine learning system:

- the attacker can only act after the training phase;
- the attacker knows only a few instances from the test set;
- the attacker only knows the output of the classifier, expressed as a probability;

TABLE 8: *Statistics obtained in the classification experiments.* MINIMIZED *and* MAXIMIZED *rows refer to the dataset modified by injecting one of the hybrid individuals; the* PUREMAXMIN *and* HYBRMAXMIN *rows refer to the dataset modified by injecting a random individual, pure or hybrid, inverting the ground truth; last column reports the* p*-value of a Mann-Whitney U Test comparing the output for instances in the modified dataset with that in the original one.*

Dataset	**Accuracy**	**P/R-AUC**	**F1**	**p-value**
ORIGINAL	0.94	0.42	0.51	(≈ 0.29)
MINIMIZED	0.95	0.29	0.41	$\approx 10^{-3}$
MAXIMIZED	0.17	0.06	0.08	$\approx 10^{-29}$
PUREMAXMIN	0.67	0.06	0.1	$\approx 10^{-15}$
HYBRMAXMIN	0.18	0.02	0.03	$\approx 10^{-26}$

TABLE 9: *Confusion matrix of the classification results obtained on the original test set, normalized over the true labels.*

Truth \ **Pred.**	NEG.	POS.
NEG.	0.95	0.05
POS.	0.24	0.76

TABLE 10: *Confusion matrices of the classification results obtained on the modified test sets, normalized over the true labels.*

Truth \ **Pred.**	NEG.	POS.
NEG.	0.95	0.05
POS.	0.24	0.76

(A) *Injection with* MINIMIZING *hybrid individuals*

Truth \ **Pred.**	NEG.	POS.
NEG.	0.13	0.87
POS.	0.01	0.99

(B) *Injection with* MAXIMISING *hybrid individuals*

Truth \ **Pred.**	NEG.	POS.
NEG.	0.68	0.32
POS.	0.49	0.51

(C) *Injection with randomly chosen* pure *individuals, inverting the ground truth (*PureMaxMin *row in Table 8)*

Truth \ **Pred.**	NEG.	POS.
NEG.	0.17	0.83
POS.	0.70	0.30

(D) *Injection with randomly chosen* hybrid *individuals, inverting the ground truth (*HybrMaxMin *row in Table 8)*

- the system is a black-box: the internal structure and the parameters are not visible;
- the system acts as an oracle for the attacker, i.e. the attacker can ask for the classification of chosen instances;
- the attacker has the goal to let the system accept vulnerable code as safe;
- the attacker can alter the input without affecting its original functionality.

With this setting, the technique allowed to successfully attack the simple chosen classifier system. Moreover, the low computational effort required to deceive the neural network, and the flexibility of the GE system in successfully achieve very good results in all the several experimental configurations listed in Section 6.3.4, suggests that the approach is promising, and that it can be applied to attack more sophisticated deep learning systems. The expectancy is that for more complex systems, or even non neural classifiers, the exploration of the input could be effective, albeit it could require different parameters that control the evolutionary algorithm.

6.4.2 *Neural Fitness Functions: Further Perspectives*

The specific considered domain, namely that of source code classification, does not allow to find adversarial examples by simply computing a gradient descent (or ascent) considering the output of the network as the loss function, with respect to a continuous input feature space [137]. This is the main challenge, compared to the more common goal of attacking image recognition systems. Therefore, all the experiments have been designed to overcome this difficulty in exploring the discrete input space of possible source code instances. The results show that this can be done by mean of an evolutionary system, guided by the output of the neural network taken as a fitness evaluator.

Notice that the main intention here lies from an adversarial perspective, but the exploitation of the neural output of a given learning system to guide an evolutionary exploration of the input space can be helpful in many domains. In general, all the knowledge learned by a system can be used as the fitness for an evolutionary system, for instance, dealing with source code, to automatically generate programs that satisfy given properties or specifications.

Further studies on the basis of these insights are discussed in the next chapters: by using a better suited grammar-based evolutionary algorithm (namely DSGE, described in Chapter 3.2), the decision process neural networks is investigated by analysing the network decision boundaries (Chapter 7) and if human concepts somehow emerge in the internal layers (Chapter 8).

7

INPUT SPACE SAMPLING AND DECISION BOUNDARIES

This chapter, which is based on the work published in [128], presents an evolutionary approach for probing the behaviour of a deep neural source code classifier by generating instances that sample its input space. First, a grammar-based genetic algorithm is applied to evolve Python functions that minimise or maximise the probability of a function to be in a certain class. Then, such sets of evolved programs are used as initial populations for an evolution strategy approach in which, by following different policies, constrained small mutations to the individuals are applied, to both explore the decision boundary of the network and to identify the features that most contribute to a particular prediction.

7.1 INTRODUCTION

In recent years, deep neural networks (DNNs), and especially neural classifiers have become popular and successful for applications in a wide range of domains. Therefore, being able to understand *how* such models make their predictions is of interest, especially with their internal architectures becoming complex and sophisticated.

As already discussed in Chapter 6, ample research studied how to generate adversarial examples for networks which classify images, that is pictures that lead them to wrong predictions. Problems arise when acting in domains where instances need to be transformed to numerical vectors to be used as input for neural networks: such transformations are, in general, not invertible, and thus classical methods based on gradient descent are not suitable. This is the case of source code, which can be given as input to neural networks only after being transformed by one of many available embedding methods. In fact, in such a case we can apply known methods for moving on the error function only with respect to the embedding input space. That is, we can for instance find the embedding vector of adversarial examples, but we would need to find a way to build the corresponding adversarial source code [67].

Here, this problem by exploring how the output of the network changes, when input source code changes. The output to be considered will not be the categorical classification, but the real valued probability that most neural models compute for each possible classification outcome. Also, the exploration aims at finding input source code leading to specific values of the probability, namely the maximal 1, the minimal 0, or the 0.5 value, which in the case of binary classification corresponds to the decision boundary of the network. The goal is to study how the solution space can be explored for finding source code snippets that produce the wanted probability values, and also

to find which small perturbations quickly change the output values. This allows to look for specific features, in the source code, which show strong correlation to the changes in the network output, in a way similar to what is done when studying the adversarial space for numerical input domains.

The proposed approach relies on moving in the source code input space by using evolutionary methods, to reach the needed network output values. In our method, individuals are source code snippets, fitness is the probability value produced by the neural classifier (similarly to what has been done in Chapter 6), and the genomes are evolved as defined by a grammar-based evolutionary algorithm. The overall mechanism involves a first phase in which individuals are generated to reach the desired fitness value (0, 1 or 0.5), and a second phase where other instances are derived from the previous ones, to investigate how – and depending on which features – the fitness changes around the reached values.

The contribution consists in the design the outlined evolutionary method, good to search the space of source code instances. Its effectiveness is demonstrated on a state of the art neural classifier [65]. Results show that in all the considered settings we obtain source code snippets inducing the required output from the neural network, and also that the system easily finds sets of derived snippets, which describe how the classification changes in a given area of the input space.

7.2 METHODS

Essentially, the approach consists in taking a neural network trained in source code classification, in using its output as the fitness evaluator for an evolutionary algorithm, and in finally looking for input programs leading it to output specific class probabilities. The idea is similar to that described in Chapter 6, but with an inherently different purpose: instead of evolving features to be injected in given program instances to produce adversarial examples able to deceive a neural vulnerability detector, here the goal is closer to the field of XAI, since it is to study which input features are the most important for the network to make its predictions.

In the literature, classifiers have been developed to predict whether input source code satisfies some given quality properties, related for instance to good software engineering practices or to avoid security flaws. In the outlined experiments, a well known and publicly available neural classifier is considered, which is based on the transformer architecture and that is trained to check some properties of Python source code snippets. Another publicly available software platform will be adapted in order to evolve individuals defined through a formal grammar in accordance to the proposed two phase process.

The behavior of the neural system will be probed by source code snippets evolved by using a simplified grammar, but they will vary

enough to discover which code features mostly affect the classification results.

7.2.1 *The CuBERT Source Code Classifier*

The benchmark source code classifier, CuBERT [65], is basically a BERT model [39] (the popular transformer for NLP, see Chapter 3.1), modified to effectively deal with source code. For the proposed experiments, three of the original fine-tuned classifiers proposed by the authors have been considered, to make the results and investigation more robust with respect to the possible bias induced by the single accounted downstream task. In particular, all the experiments have been replicated on three fine-tuned models[1] trained on the following binary classification tasks:

VARIABLE MISUSE this task is referred to the mistakes developers could make when dealing with similar code fragments or similar variable names (e.g. when copy-paste code snippets but forget to properly renaming variables) [5]. In this fine-tuned model, it consists in detecting if there is a variable misuse at any location in the Python function given as input;

WRONG BINARY OPERATOR this task [112] simply consists in detecting weather there is a binary operator that is improperly used in an expression occurring in the function given as input;

SWAPPED OPERANDS a task that consists in detecting if there are operands of non-commutative binary operators that are swapped with respect to the correct usage intention.

The focus, in this work, is to study how different syntactical features mostly influence the evaluation of the different models, under the hypothesis that the relevance of such features is related to the problem to solve. It should be remarked that, to this end, the main interest lies in the numerical variation of the output instead of the prediction in terms of classification decision. This is due to the fact that a ground truth for the described problems is difficult to establish for programs that do not have a predetermined functionality, and thus the focus won't be on the creation of adversarial examples, but only on evolving programs that lead to arbitrary predictions, no matter their *correct* label.

7.3 TWO-STAGE EVOLUTIONARY SEARCH

In this approach, the exploration of the CuBERT behaviour is performed in two main phases: in the first one, the CuBERT solution space is sampled by applying pure DSGE (see Chapter 3.2 for a full working explanation) using the simplified Python grammar reported in Figure 27 and different fitness objectives; in the second one, only

1 https://github.com/google-research/google-research/tree/master/cubert

```
<start>         ::= def <funid>(x, y):
                    {:<statements>:}
<funid>         ::= funid | sum_xy | sub_xy
<statements>    ::= <statement>\n
                    | <statement>\n<statements>
<statement>     ::= <varid>=<simpl-expr>
                    | return <simpl-expr>
                    | <if-stmt>
                    | while <atom><condoperator><atom>:
                      {:<statements>:}
<simpl-expr>    ::= <atom> | <atom><operator><atom>
<atom>          ::= <varid> | <int>
<if-stmt>       ::= if <atom><condoperator><atom>:
                    {:<statements>:}
                    | if <atom><condoperator><atom>:
                      {:<statements>:}
                      else:
                      {:<statements>:}
<int>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<varid>         ::= x | y | sum_ | sub_ | mult_ | mod_ | div_ | rem_
<operator>      ::= + | - | * | / | % | **
<condoperator>  ::= == | != | >= | <= | < | >
```

FIGURE 27: *Simplified Python grammar.*

"constrained" mutations are applied to the evolved individuals, to point out how such mutations affect the fitness.

7.3.1 *DSGE and Neural Fitness Functions*

In this first experimental phase, an evolutionary algorithm based on DSGE is applied for sampling the solution space of the fine-tuned CuBERT models described in Chapter 3.1.

Basically, in DSGE, genotypes are represented as lists of ordered derivation steps of a given CFG: each genotype is a sequence $S_1, \ldots, S_n$, where for each $i \in \{1, \ldots, n\}$, S_i is referred to a non-terminal symbol of a context-free grammar $G = (N, T, A, P)$ where N and T are the sets of non-terminal and terminal symbols, respectively, $A \in N$ is the axiom, or the starting symbol of G, and P is the set of production rules. Each S_i is an ordered list of integers $r_1, \ldots, r_k$, where k is the number of times the i-th non-terminal is expanded, and the values correspond to the indices (in the grammar) of the applied expansion rules. For a complete explanation of DSGE, refer to Chapter 3.2.

In the three considered fine-tuned CuBERT models, which are binary classifiers, the output is a value $0 \leqslant v \leqslant 1$, and an individual is predicted to be in the negative class if $v \leqslant 0.5$ and in the positive class otherwise. Given this premise, using the CuBERT output as the fitness, for each fine-tuned model programs are evolved with DSGE to pursue three fitness objectives: maximization, minimization, and minimization of $|0.5 - v|$. The founding idea is to produce pools of individuals that the network classifies with an high confidence (namely the program instances whose fitness is very close to 0 or to 1) or for which the prediction is uncertain, that is the individuals whose fitness is close to the decision boundary 0.5.

In this experimental phase the evolved programs are compliant with the grammar listed in Figure 27. The grammar formalizes an

extremely minimal Python function definition in which two variables `x` and `y` are fixed parameters, the only control flow constructs are `if`, `if-else`, and `while` statements, and only assignments and operations on integers are valid instructions. Note however that the choice of a so simplified Python grammar, which is made for readability reasons, allows to effectively produce individuals yielding outputs covering all the interval between 0 and 1, and thus it suffices for this discussion. Finally, notice that since the interest here is to analyse the output changes by varying syntactical elements, the evolved programs are not supposed, in general, to be executable or meaningful.

7.3.2 *Mutations and Fitness Variations*

This second phase is aimed at exploring how the fitness of individuals changes when different mutations are applied, to identify which features are most salient for the network to make its prediction.

In DSGE, the standard mutation operator is defined, for a position $S_i = r_1, \ldots, r_k$ as a random change of an integer r_j into another valid integer r_j^* for that gene. In other words, a mutation is basically the selection of a different expansion option for a non-terminal symbol. It should be noted that, in general, r_j^* could be referred to a production rule that contains non-terminal symbols of the grammar. In that case, for each non-terminal, random expansions rules are selected (and thus added to the genome) until all the non-terminals are resolved into terminal symbols. Given this definition, it is possible to specify a different mutation operator in which only some positions can be altered, meaning that only some non-terminals can be changed in their expansion rules.

By means of two constrained mutation operators (whose pseudocode are reported in Algorithm 3), the fitness variation is explored to determine how it is related to the non-terminals for which the mutation is enabled, to identify which syntactical features are the most salient for the network prediction. To this end, two families of experiments have been performed:

1. The exploration of the neighborhood of individuals. For this investigation, mutations are applied by following the procedure described by Algorithm 4 and then the average fitness variation of individuals generated by the mutations is measured;

2. following a $(\mu + \lambda)$ evolution strategy approach [94] (i.e. a population based algorithm in which the best μ individuals of a population are mutated λ/μ times for composing the new generation), the extent of how the different constraints affect the fitness variation along a sequence of mutation steps is measured. This procedure is formally described by Algorithm 5.

The core difference between the two experimental investigations is that, while in the first an extensive exploration of the neighborhood of the individuals is performed, in the second one a similar exploration is conducted over many mutation steps guided by an evolu-

Algorithm 3 Constrained mutation operators

ind ← individual to mutate
pmut ← mutation probability
G ← list of genes $g_1, \ldots, g_n$
▷ each $g_i = v_1, \ldots, v_m$ is associated to a non-terminal, occurring m times in the sentential form, and each v_j specifies the grammatical rule used in each position

▷ The difference between the two operators is that, while in CNSTR_MUTATE_1 each mutable position is mutated with probability *pmut*, in CNSTR_MUTATE_2 a mutation in a gene occurs with probability *pmut* and in that gene only one random position is mutated

procedure CNSTR_MUTATE_1(*ind*, *pmut*, G)
    for all $g \in G$ **do**
        $V \leftarrow$ values in the genome for the gene g
        for all $v \in V$ **do**
            if RANDOM() $\leqslant$ *pmut* **then**
                mutate v into another valid integer v^*
                if rule v^* contains non-terminals **then**
                    randomly expand all the non-terminals
                end if
            end if
        end for
    end for
    return *ind*
end procedure

procedure CNSTR_MUTATE_2(*ind*, *pmut*, G)
    for all $g \in G$ **do**
        if RANDOM() $\leqslant$ *pmut* **then**
            $V \leftarrow$ values in the genome for the gene g
            randomly choose $v \in V$
            mutate v into another valid integer v^*
            if rule v^* contains non-terminals **then**
                randomly expand all the non-terminals
            end if
        end if
    end for
    return *ind*
end procedure

tionary pressure and controlled by the fitness function. In the first case, the considerations that follow from the results, which will be fully reported and discussed in Section 7.4, are mainly quantitative: it can simply be observed how the different constraints in the mutation operator change the fitness evaluation of the mutated individuals when compared to the fitness of the original individual. In the second experiment, indeed, the attempt is more qualitative: starting from a single individual, and by mutating (almost) a single token in each step, the features that most influence the prediction of the network can be detected, by observing how the fitness varies along many mutation steps.

Algorithm 4 Neighborhood exploration

```
P ← initial population
G ← list of indices of the mutable genes
n ← number of neighbours
E ← empty list                                   ▷ list of errors

for all p ∈ P do
    f_p ← FITNESS(p)
    for i = 1, ..., n do
        N ← empty list
        newInd ← CNSTR_MUTATE_1(p, 0.5, G)
        append newInd to N
    end for
    for all ind ∈ N do
        f_ind ← FITNESS(ind)
        append |f_i − f_ind| to E
    end for
end for

output the mean of the values in E
```

7.4 RESULTS

This section we supplies the technical details of the performed investigations, and discusses the obtained results. All the experiments have been performed on a Linux machine with 16GB RAM, 4 CPUs running at 3.60GHz and a Nvidia GTX 1070 GPU, with tensorflow over CUDA, and sge3 implementation[2] of DSGE.

7.4.1 *Sampling the solution space*

As described in Section 7.3, in the first experimental phase DSGE is applied for evolving Python programs using, as the fitness function, the outputs of the three CuBERT fine-tuned models described in

2 https://github.com/nunolourenco/sge3

Algorithm 5 Fitness variation over mutation steps

```
P ← p_1, ..., p_λ                                   ▷ initial population
G ← list of mutable genes
μ ← number of parents
n ← number of mutation steps

for n times do
    sort P according to the fitness of each p_i ∈ P
    B ← p_1, ..., p_μ                              ▷ list of best μ individuals
    P ← empty list
    for all p ∈ B do
        append p to P
        for λ/μ times do
            m ← CNSTR_MUTATE_2(p, 1/|G|, G)
            append m to P
        end for
    end for
end for
```

Section 7.2.1, namely the three binary classifiers trained in detecting variable misuse, swapped operands and wrong binary operators.

For each of these models, three fitness objectives have been considered (i.e. minimization, maximization and minimization of the distance between 0.5 and the output) and for each of these combinations 10 DSGE runs have been performed, by using as the reference grammar the one shown in Figure 27. For each run, populations of 50 individuals have been evolved for 50 generations with tournament selection (3 individuals per tournament), keeping an elite of 10 individuals at each generation, and by letting crossover and mutation occur with probability 0.9 and 0.1, respectively. Also, the size of individuals have been limited by imposing a maximum tree depth of 25.

The obtained results, which are fully reported in Figure 28, show how it is always possible to pursue the fitness objective, even if using an extremely simple and minimal grammar. For each model, the targeted fitness that seems to require more generations to be reached is 0.5. This value is indeed the most interesting for the scope of this work, since it represents the decision boundary, that is the classification threshold of the network.

7.4.2 *Moving across decision boundaries*

This section discusses how different syntactical features affect the prediction of the network. Specifically, with the two constrained mutation operators detailed in Algorithm 3, and by applying them to the individuals evolved with DSGE by following different policies, it is possible to observe and study the fitness variations.

The first investigation, formally outlined in Algorithm 4, is aimed at a quantitative assessment of how the different mutations affect the predictions of the three considered CuBERT models. For each

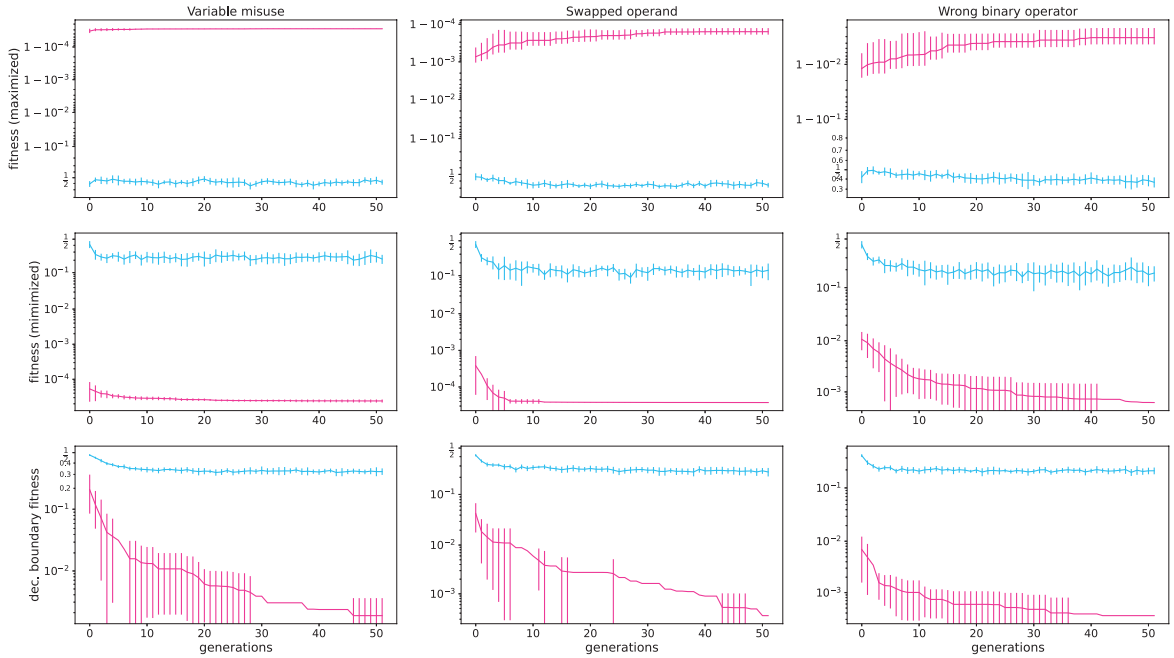


FIGURE 28: *Plots of the fitness obtained with DSGE for the considered models and fitness objectives. Blue lines represent the average population fitness, purple lines the fitness of the best individual. Vertical bars report the variance in over the different runs.*

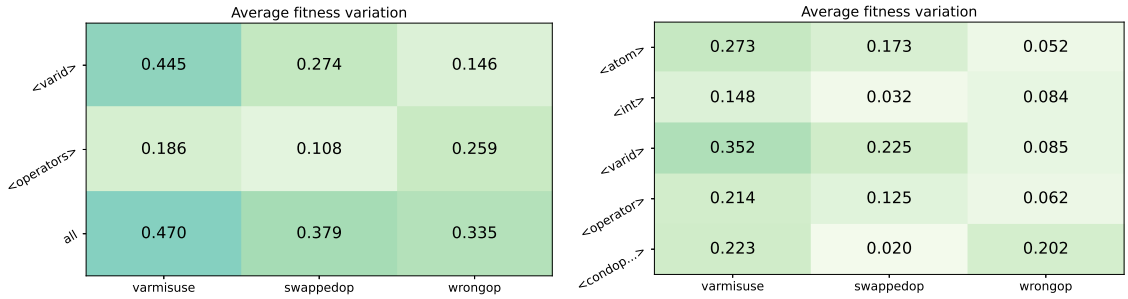
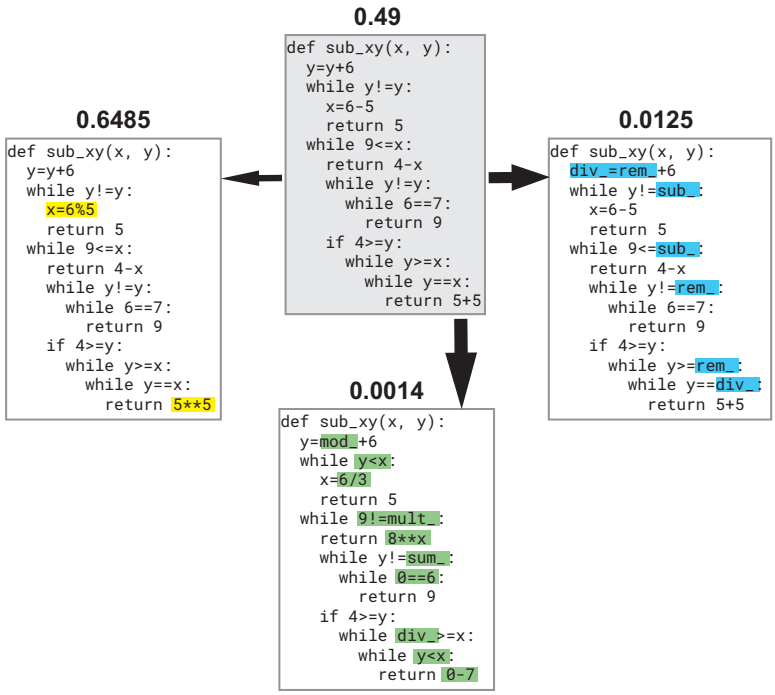


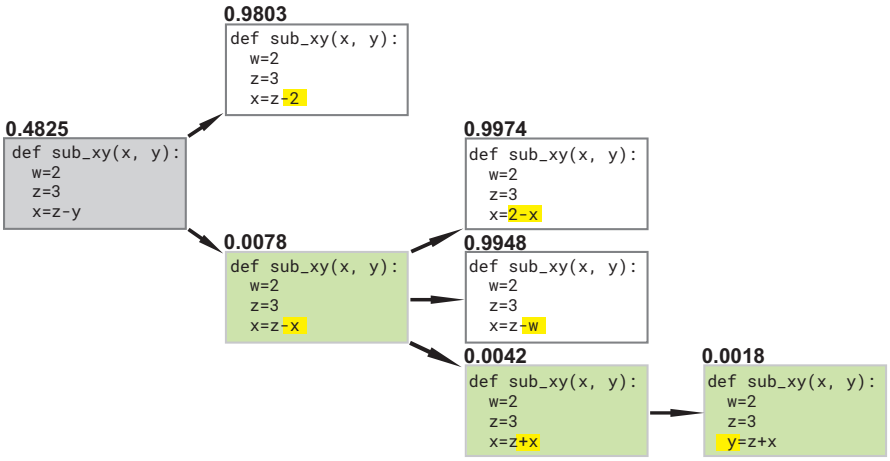
FIGURE 29: *Results of the mutation-based experimental phase. Average fitness variation for the three fine-tuned models when different constraints on the mutation operator are imposed and the neighborhood is explored with a single mutation step (left) or several steps (right).*

model, the starting points are the individuals evolved in the previous phase when pursuing the fitness objective near the decision boundary, namely a set of 10 evolved individuals for which the networks outputs a value very close to 0.5. Then, starting from each of individual, 10 neighbours are generated by applying the CNSTR_MUTATE_1 operator, by setting the mutation probability to 0.5 and by imposing three different constraints:

1. mutation allowed only for variable identifiers, that is the only mutable gene is that related to the `<varid>` non-terminal;
2. mutation allowed only for the arithmetic operators, that is the only mutable gene is that related to the `<operator>` non-terminal;
3. a less tied constraints set, in which mutation is allowed for the genes related to `<int>`, `<varid>`, `<operator>` and `<condoperator>`



(A) *Neighborhood exploration: single mutation step where several mutations are allowed.*



(B) *Evolution strategy: several mutation steps with one mutation allowed for each step. Green backgrounds highlight a "winning" path.*

FIGURE 30: *Examples for the mutation-based experimental phase. The numbers on top of the boxes represent the fitness value.*

non-terminals. These constrains are labeled as "all" in Figure 29 (left).

Notice that all these constraints imposed for the mutation operator are related to non-terminal symbols whose expansion rules lead to terminal symbols, meaning that each mutation do not alter the syntactical structure but takes action only on the AST leaves. Then, for each possible pair of constraints and task, the average fitness variation of the mutated individuals has been measured, as reported in Figure 29 in the left diagram. Figure 30 shows an example of this neighborhood exploration, along with a graphical representation of the mutation paths studied in the experiments described in Section 7.4.3.

7.4.3 *Blind Spots and Salient Features*

This last set of experiments, formalized in Algorithm 5, lets the evolutionary machine free to explore how to improve the fitness of individuals, by changing at most one derivation from each occurring non-terminal. As in the experiments described in the previous section, only a chosen subset of non-terminals were mutable. Also, such exploration spanned a sequence of at most 5 mutation steps.

In analogy to what can be done when exploring a geometrical space, this evolutionary search moved from a parent individual to a new one by following a specific direction, here represented by a mutation involving a given grammar rule. For instance, by means of the grammar in Figure 27, the evolutionary exploration moved from the individuals evolved in the first phase, to new individuals obtained by mutating (for 5 generations) their DSGE representation in terms of choices made when applying rules of non-terminals `<atom>`, `<int>`, `<varid>`, `<operator>`, and `<condoperator>`. The details of how such mutation operator works are given in CNSTR_MUTATE_2 of Algorithm 3. For each generation, best individuals have been selected as described in Algorithm 5, following an evolution strategy $(\mu + \lambda)$ with $\lambda = 12$ and $\mu = 3$. The optimization goal of this phase, starting from individuals sited near the decision boundary, has been either to maximise or to minimise the fitness. This turns into asking the system to find individuals that move far from the decision boundary and across the two sides it separates, always looking at the network's behavioural changes on the individuals belonging to the new generation.

Two main resulting sets of data have been examined: the fitness values that can be reached, and the grammar rules that are more effective in varying the fitness value. These data offer a view on the behavior of the network, for instance by telling us which syntactic elements the network is more sensitive to, and when they impact more than others on varying the fitness value, namely the probability that the network assigns to instances for belonging to a certain class. The results of the evolutionary exploration of the input space, around instances classified close to the decision boundary, have been infor-

mative with respect to the assessment of the tested neural network. The evidences can be outlined as follows:

- even when the evolution applied to the starting individuals did not modify their real class with respect to the chosen tasks, as guaranteed by the set of changes allowed in the derivations, the evolutionary system is always able to derive individuals with fitness close to the desired value, 1 or 0, meaning that we can find adversarial examples for the network;
- each task was a different challenge for the evolutionary system, that, for instance require more steps to reach the optimal fitness when checking for swapped operands than for variable misuse, or that forced the evolution of longer individuals when aiming to fitness close to 0, than for the opposite goal;
- for each of the three classification tasks, the networks showed higher sensitiveness for specific sets of non-terminals as it can be seen in Figure 29 (right), in details: `<atom>` and `<varid>` for variable misuse, `<atom>` and `<varid>` for swapped operand, with overall impact from the other non-terminals different from what happens with variable misuse, `<condoperator>` for wrong operator.

The last remarks, on which non-terminals induced the fastest variation of fitness for a single evolutionary step, deserve some considerations, in relation to the understanding of the actual behavior of the network. In the experiments, it is possible to see how the classifier reacts to small, and specific, variations in the input it receives. This information can be compared to what it is expected from the trained network, with respect to the task at hand. In this setting, for instance, we can see that even if the system were allowed to evolve individuals by changing integer constants (being allowed to mutate the use of the `<int>` non-terminal), this almost never modified the classification an any of the three original tasks, and this seems to be correct. Also, having the network of the variable misuse task impacted by changes related to the `<varid>` non-terminal is expected. On the other hand, we discovered that the network for the wrong operator task is more sensitive to changes in the choice of conditional operators, than in changes among arithmetical operators (`<condoperator>` and `<operator>` non-terminals). This was unexpected, and it could perhaps point to a bias in the training process.

Finally, an interesting visualization of what the system allows to describe, concerning how the classification of the network changes when moving through instances in the space around the decision boundary is shown in Figure 30b, where it is possible to follow the progression of an individual that starts with fitness equals to 0.5 and ends with a fitness close to 0, during the mutation steps of the second experimental phase.

7.5 DISCUSSION

With the outlined grammar-based evolutionary approach, we are able to search the input space of a neural network looking for instances that lead to arbitrary probability predictions. Also, we can explore their neighborhood and look for salient variations in the input-output mapping that characterizes the classifier. The method has been tested on a state of the art source code neural classifier, namely the CuBERT transformer, and allowed to identify which syntactical features of the source code mostly impact the classification. This way to probe the behavior of a network, in wide input space areas, can be used to look for adversarial examples, but also to derive deeper information about the sensitiveness of the classifier with respect to features of input instances.

To further this line of research, this approach can be applied to check the robustness of defense proposals in the area of adversarial attacks, e.g. in setting similar to that described in Chapter 6, but from a defensive perspective. A broader application area will be that of neural network understandability, for any neural model and also under a black-box approach, only assuming to have access to the predicted class probabilities. Searching the input space for instances located in key areas, with respect to the neural model decisions, could give insight to what the classifier is actually taking as key feature of the input, or to where it has blind spots or distorted evaluation of the source code snippets. A similar intent drives the work described in Chapter 8, where the same network architecture is inspected to check which human concepts are significant to make the prediction.

8

CONCEPT-BASED EXPLAINABILITY

This chapter is aimed at investigating the decision process of neural networks. The founding interest is similar to that described in Chapter 7, but here the problem is addressed from a different perspective: instead of looking for syntactic features that affect the predictions of a source code classifier, here the focus is on human-defined concepts. The study is spread along two experimental directions: in the first one, we study the activations of the neurons of a transformer trained in the detection of software vulnerabilities so as to identify if (and, possibly, where) some human understandable concepts emerge in the network. In the second one, we generate programs by applying a grammar-based evolutionary algorithm with a fitness function that favours individuals which stimulate (or weaken) the activations in neurons where given concepts majorly emerge. Finally, we study how the output of the network varies on sets of evolved programs, to assess how the evolutionary pressure along the direction of a concept affects the prediction.

The reported work has been previously published in [129] in a preliminary version, and then extended in [51].

8.1 MOTIVATIONS AND WORK OVERVIEW

Explaining *how* machine learning models, and specifically deep neural models make their predictions is an intriguing and valuable research issue, especially in recent years, with deep models becoming popular in many areas, and with their architectures becoming increasingly complex.

This work plays in this direction, aiming to find within the internal layers of a deep neural transformer trained in the detection of software vulnerabilities the emergence of human understandable concepts, such as the lack of checks on user input or the use of unsafe functions. The goal is to both explore if what is relevant for the network to take a decision in a given context (e.g. that of cybersecurity) is somehow comparable with what is important for a human being in the same context, and to identify possible blind spots or misconceptions in what the network has learned. In this field, evolutionary techniques seem to be be very promising (see e.g. Chapter 6 and 7), since they allow to generate possible inputs for a network under an evolutionary pressure. The chosen fitness function drives the production of individuals maximising (or minimising) the neural outputs, namely the activations yielded in a given internal region of the network. More in details, the contributions are the following:

- the adjustment of an earlier approach [71], in terms of applicability to the transformer architecture and with a different choice

for linear separator on the activations space, for identifying if some human comprehensible concepts emerge in the internal layers of a deep neural network;

- the validation of the approach on a CuBERT transformer model (described in Chapter 3.1) trained in the detection of software vulnerabilities;
- a grammar-based evolutionary algorithm designed for evolving programs that mostly stimulate the regions in the network where arbitrary concepts majorly emerge, by means of a fitness function defined as the distance from an hyperplane that, according to the presence of a given concept, separates input instances when seen as points in a space defined over the neural activations;
- a study of the classification performance of the model on the evolved programs, to show how some concepts are relevant for the network, i.e. how the presence, in the input, of given concepts affects the output.

8.2 APPROACH DESCRIPTION

The proposed approach is devised at studying the decision process of a neural classifier. The experimental workflow, which is outlined in Figure 31, basically consists in treating the input instances of a neural network as points of the geometric space defined in the internal layers of the network, as it will be detailed in Section 8.2.2. Together, within the input dataset, samples of instances that represent a given concept are selected. In general, whatever is expressible in natural language can be considered as a concept. Here, in the specific domain of source code processing, and being the considered dataset composed by Java methods, both lexical concepts (e.g. the presence of targeted patterns or constructs), and syntactical ones, such as the complexity or the relation between the types of the arguments and the returned objects will be considered. Thus, it is possible to find an hyperplane in the activations space that separates (with some approximation) the points that represent a concept from those that do not represent it. With this equipment one can finally consider the perpendicular direction with respect to the hyperplane as the direction representing the concept, and eventually use the signed distance from the hyperplane as a fitness function for an evolutionary algorithm to evolve programs representing a given concept.

8.2.1 *Input instances and sub-concepts*

One of the crucial problems in explaining the decisions of machine learning models, is that they usually operate on input features (e.g. matrices of pixels or sequences of textual tokens) that are difficult to be interpreted when examined by a human being. To this end, in this work, instead of operating directly on features [45], the focus is

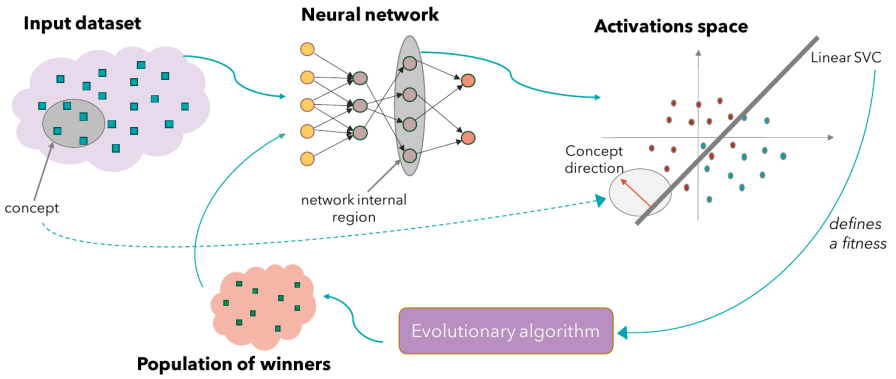


FIGURE 31: *Detection of concepts and evolutionary input space exploration guided by the emerging concepts.*

moved at an higher level and reason on concepts that are meaningful for humans.

The deep neural model considered in this work, namely the CuBERT transformer [65], that has been previously described in Chapter 3.1, is internally composed by 24 encoding blocks, each having a final feedforward layer. Thus, for each $l = 1, \ldots, 24$, the output of the feedforward layer l can be expressed as a function $f_l : \mathbb{R}^n \to \mathbb{R}^{n \times m}$, where n is the dimension of the input sequences accepted by the model, and m is the number of neurons in the layer. Notice that, as specified in this notation, each neuron has a vectorial output (rather than a scalar one) with n dimensions, that is the same dimensionality of the input vector. In this work, feedforward layers with $m = 1024$ neurons, and input dimension $n = 512$ will be always considered.

In this work, the main goal is to study which human comprehensible sub-concepts (with respect to the original task the model is trained on) are relevant for the decision process of the network, and to investigate if such emerging sub-concepts are similar to those considered by a human being when addressing the same task. For instance, if the main task is the detection of zebras in images, a possible sub-concept can be the presence of stripes in the image being tested. Similarly, in the source code analysis domain, a sub-concept related to the detection of a vulnerability like the classic buffer overflow [108] could be the presence of calls to unsafe library functions such as `strcpy()`.

8.2.2 *Activations space, linear SVCs and concept-based neural fitness function*

More precisely, given a sub-concept c, and a set S of possible input instances, we can express c by partitioning S into two subsets P_c and N_c containing, respectively, the positive and negative instances with respect to the concept c. In [71], concept activation vectors (CAVs) were used to explore the ability of layers of the network when employed to separate input instances belonging to P_c or to N_c. Here a similar approach is devised: the vector space obtained by concatenating the activations of all the neurons of a layer is considered, and a linear support vector classifier (SVC) [19] is trained to correctly rec-

ognize the activations yielded by the instances belonging to P_c from those belonging to N_c.

The rationality in chosing SVCs and linear kernels as simple concept classifiers, working on the layer activations generated from specific classes of input instances, is inspired by that found on a previous work [130], which compares activations of deep neural networks to signals from the brain, when stimulated with some sensorial input. Formally, given a feedforward layer l, for each $s \in S$, we compute $f_l(s) \in \mathbb{R}^{n \times m}$, and we flatten its rows by applying a transformation $f_l^* \colon \mathbb{R}^{n \times m} \to \mathbb{R}^t$, where $t = nm$ and $f_l(s)_{x,y} = f_l^*(s)_{nx+y-1}$, for all $x \in \{1, \ldots, n\}$ and $y \in \{1, \ldots, m\}$. Then, we define the binary ground truth for the concept c by means of the two sets $S_1 = \{f_l^*(s) \colon s \in P_c\}$ and $S_0 = \{f_l^*(s) \colon s \in N_c\}$; finally, a linear SVC can be trained on those activations vectors. This binary classifier $V_{c,l}$, corresponding to a separating hyperplane for the activations space of layer l, can then be used to determine the direction of the concept c. By definition, a linear SVC $V_{c,l}$ separates the instances with respect to the sub-concept c, by means of a decision function $d_{c,l} \colon \mathbb{R}^t \to \mathbb{R}$ that represents the signed distance between the hyperplane $V_{c,l}$ and the point determined by the activations yielded by the instances in the layer l. Specifically, since here only binary classification problems are considered, the SVC predicts an instance s to be in class 0 or in class 1 depending on whether $d_{c,l}(f_l^*(s))$ assumes a negative or a positive value. In this way, an instance s is predicted to represent or not to represent a sub-concept c according to where, in the space, the activations generated in layer l are placed with respect of the separating hyperplane $V_{c,l}$.

Given this premise, the fitness function for an evolutionary algorithm can be defined as the signed distance $d_{c,l}(s)$ computed by feeding the network with the individual s and by considering the SVC $V_{c,l}$ and the corresponding point in the activations space of the feedforward layer l. Since high-level programming languages usually build source code programs under the control of a context-free grammar, in the scope of this chapter the idea is to apply a grammar-based genetic algorithm, in order to evolve programs according to a fitness function that represents the direction of a concept. To this end, DSGE [91] is the easy choice, since it allows to efficiently evolve individuals that comply with a given formal grammar. As shown in Figure 3 on page 21, which reports an example of the decoding process of genotype into a phenotype on the basis of a simple BNF grammar, in DSGE the genotypes are represented by sequences $G_1, \ldots, G_n$, each listing derivation steps of a grammar. At each position k, G_k is referred to a non-terminal symbol of the given context-free grammar, and it is made of a list of integers whose length represents the number of times the k-th non-terminal is expanded, while the values represent, for each expansion, the index in the grammar of the applied rule for that non-terminal. It should be remarked that in DSGE the evolution operates on genotypes; in all the proposed experiments, at each generation, the genotypes are decoded into the corresponding phenotypes, and the fitness is computed on them.

8.2.3 *Sensitivity to sub-concepts*

With the outlined setting, it is now possible to measure how much the presence (or the absence) of a concept affects the decision of the network on the task it is trained on. To this end, two approaches have been addressed:

1. The first is similar to the original TCAV [71], which uses directional derivatives to compute the conceptual sensitivity on entire classes of inputs;
2. The second is based on the evolutionary synthesis of classes of inputs and on a subsequent test of the classification performance on these evolved sets.

While the first point is basically used here as a benchmark, the second is expected to be more expressive and capable to widely explore the input space and to look for possible blind spots or misconceptions. For each concept c, the fitness function for DSGE is computed by considering the layer l for which the support vector $V_{c,l}$ obtains the highest accuracy. Two directions are devised:

- By using a simplified Java grammar (Figure 32) sets of individuals that *maximise* the fitness are generated, and on these sets the number of instances that are classified as vulnerable is counted.
- By using a grammar that forces the presence of the vulnerability the model is trained on, sets of individuals that *minimise* the fitness are generated, and on these sets the number of instances that are classified as vulnerable is counted.

Although the two points are similar in their design, they are conceived to address different issues. The maximization experiments are intended to find out the concepts that positively affect the classification. Here, the ground truth of all the evolved programs is assumed to be negative, since the grammar used always produces programs that are syntactically correct but semantically meaningless, and furthermore not necessarily compilable or executable. In the minimization experiments, instead, the ground truth is assumed to be positive since, even though the grammar suffers from the same problems, it introduces lines of code that represent the vulnerability. These experiments are intended to detect which concepts lead to a misclassification when they are counteracted.

8.3 EXPERIMENTS

The performed experiments essentially consist in in the investigation of possible sub-concepts that emerge in the internal layers of a deep neural network trained in the classification of Java source code, and then in the application of an evolutionary algorithm able to evolve Java programs using a fitness function derived from these concepts,

```
<start> ::= public <retype> <funid>(){\n<statements><return-stmt>;}
          | public <retype> <funid>(<parameters>){\n<statements><return-stmt>;}

<parameters> ::= <parameter>
               | <parameter>, <parameter>
               | <parameter>, <parameter>, <parameter>

<parameter> ::= int <varid> | int[] <varid>

<statements> ::= <statement>;\n | <statement>;\n<statements>

<if-stmt> ::= if(<condition>) {\n<statements>}
            |if(<condition>) {\n<statements>} else {\n<statements>}

<condition> ::= <atom> == <atom> | <atom> != <atom> | true | false

<while-stmt> ::= while(<condition>) {\n<statements>}
               | do {\n<statements>} while(<condition>);

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9          <retype> ::= int | int[] | void

<int> ::= 0 | <digit> | <digit><int>                    <varid> ::= x | y | z

<return-stmt> ::= return | return <simpl-expr>          <funid> ::= funid | good | bad

<simpl-expr> ::= <atom> | <atom> <operator> <atom>     <sigint> ::= <int> | (-<int>)

<atom> ::= <varid> | <sigint> | <varid> [<int>]        <operator> ::= + | -

<statement> ::= <varinit>                      <varinit> ::= int <varid>
               | <varid>=<simpl-expr>                      | int[] <varid>
               | <varid>[<int>]=<simpl-expr>               | int <varid> = <simpl-expr>
               | (int) <varid>=<simpl-expr>                | int[] <varid> = new int[<int>]
               | <if-stmt>
               | <while-stmt>
```

FIGURE 32: *Simplified Java grammar that allows the emergence of the considered possible sub-concepts.*

with the twofold objective of studying which elements mostly influence the decision of the network, and how such elements can be eventually used to deceive its prediction.

8.3.1 *Data Preparation and Fine-tuning*

The model chosen for the experiments, as in the work described in Chapter 7 is the CuBERT [65] transformer, whose architecture has been already described in Chapter 3.1. In this work, to appoint the cybersecurity domain, a CuBERT model pre-trained on a Java corpus[1] has been fine-tuned in the detection of software vulnerabilities. To this end, the Juliet Test Suite v1.3 for Java [17, 18] has been used. For each CWE, the dataset consists of a set of Java files defining both flawed methods and, correspondingly, a number of non-flawed constructs. To build the training sets, the files labelled as CWE-369 and CWE-789 – representing the "divide by zero" and "memory allocation with excessive size value" vulnerabilities, respectively – have been preprocessed. The set of positive instances was populated with the methods that self-contain the flaw, in fact all the methods that to exhibit the vulnerability need to call other methods or to use support classes have been removed.Moreover, the set excluded all the methods whose tokenization yields vectors having more than 512 dimensions, the maximum input dimension for the CuBERT model, so as to avoid truncation of the input instances. Finally, for the set of potential positive instances all the non-flawed methods labelled with

1 Available: https://github.com/google-research/google-research/tree/master/cubert, version of July 11, 2021

any CWE except CWE-369 and CWE-789, from the Juliet dataset, have been considered.

8.3.2 *Sub-concepts formulation*

For the experiments, both sub-concepts that are expected to be relevant for the detection of the considered vulnerabilities (e.g. the presence of a cast to integer), and generic sub-concepts that can be identified in the source code, but that are supposed not to be related to the vulnerabilities (e.g. the cyclomatic complexity) have been taken into account. In details, the following sub-concepts have been tested:

CAST TO INTEGER The presence (or absence) of a cast to integer operation. This sub-concept can be significant when dealing with both the divide by zero and uncontrolled memory allocation vulnerabilities;

SQUARE BRACKETS The presence of an high number (i.e. $>= 12$) of square brackets. This concept can be relevant in general, since it is strictly related to the presence of an high number of accesses to array elements;

CYCLOMATIC COMPLEXITY This sub-concept [97] addresses the structural complexity of a program, and it is a classical software engineering metric (see Section 5.4.1 for a brief description). Here, having cyclomatic complexity higher than 10 has been considered as a concept;

I/O RELATIONSHIP This sub-concept considers the *semantic* of a method in terms of the relation between what is passed as argument (i.e. the input) and the returned object (i.e. the output). Only a subset of all the possible I/O relations has been considered, namely the presence or the absence of an array among the input arguments and whether the returned object is an array or a single element. In particular, the treated sub-concept was the many-to-many relation, namely the methods that contain (at least) an array among their arguments, and that return an array.

RANDOM This concept is defined by simply assigning random labels to the methods in the dataset. The obtained partition is obviously meaningless, and the experiments on this concept are used as a baseline to assess the validity of the other results.

Excluding the baseline random concept, the first two sub-concepts resemble some traits that a human expert takes into account when attempting to identify the presence of vulnerabilities, while the others represent elements that are relevant and recognisable, but that are not directly related to the vulnerabilities in account. As an analogy in the image processing domain, for a model trained in recognising zebras we can think of concepts like the presence of stripes or having four paws as sub-concepts related to the task, and prevalence of a given color as unrelated concept.

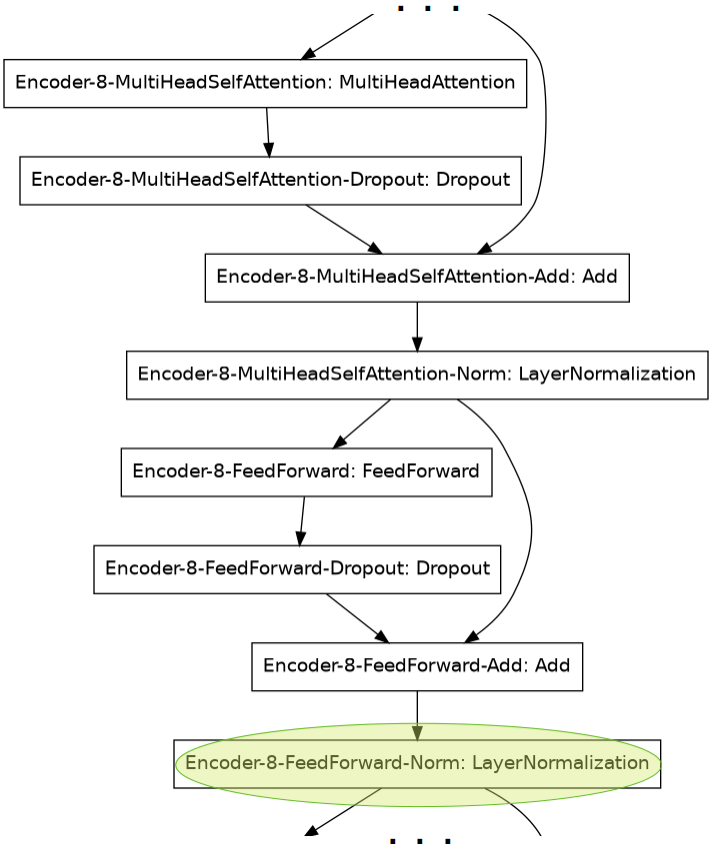


FIGURE 33: *An encoder block of the CuBERT transformer. The activations spaces are defined on the highlighted feedforward layers.*

8.3.3 *SVCs and activations spaces*

As outlined in Figure 33, the feedforward layers of the encoder blocks have been accounted, namely the dense layers that feed the succeeding encoder block. In the CuBERT transformer there are 24 encoding blocks, each one composed by 1024 neurons with a 512-dimensional output. For each considered sub-concept, first the activations obtained by flattening the output of the feedforward layers (see Section 8.2.2) for a balanced random sample have been collected, where 100 program instances were associated to the concept, and 100 did not represent it. Then, for both the fine-tuned models (i.e. the binary classifiers trained in the detection of the divide by zero and uncontrolled memory allocation vulnerabilities, respectively), and for each concept, the activations of the flattened layers have been gathered as points of a $512 * 1024$-dimensional vector space, and for each of the 24 layers a linear support vector classifier (SVC) has been trained to correctly separate the points representing programs related to the chosen concept, from the points representing programs with no presence of the same concept.

Ad the end of this procedure, besides the average accuracies obtained by the SVC over 10 runs, which are reported in Figures 35 and 36, for each concept the "best" layer (i.e. the layer that can most fruitfully used to distinguish programs that hold the concept from programs that do not, by considering its activations values) had been

identified and the corresponding best SVC. The latter will be chosen as fitness evaluator for an evolutionary algorithm, as detailed in the upcoming Section 8.3.4.

8.3.4 *Evolutionary search along sub-concepts directions*

Given the two transformers, fine-tuned respectively to recognise CWE-369 and CWE-789, it is possible to analyse how their accuracy is influenced when instances have a strong or weak presence of a given concept, to discover possible biases or misconceptions in in what the networks have learned. The method is based on building source code instances with DSGE by means of a Java grammar (Figure 32), which is simplified but flexible enough to produce the syntactical elements required by both the specific concepts and the specific classification tasks. Moreover, during the evolutionary search one can also force the grammar to always generate individuals which contain vulnerabilities for a given CWE. This is obtained by modifying the grammar with the addition of production rules expanding into vulnerable constructs which the DSGE cannot rule out.

Here, the DSGE fitness function is exactly the decision function of a chosen SVC, namely the SVC that best separates the instances with respect to a given concept, as detailed in Section 8.2.2. The decision function takes as input the activations vector of the layer, which is generated by feeding the network with the source code of an evolved individual. Given that the chosen SVC is the one that reaches the highest accuracy for the concept under investigation, among the ones that are built for all the feedforward layers in the network, the fitness values will be higher when the presence of the concept is stronger in the input source code, and lower when such presence is weaker.

The DSGE evolutionary process can thus be defined through the following settings:

- the CONCEPT, among the five described in Section 8.3.2;
- the considered CWE, between CWE-369 "divide by zero" and CWE-789 "memory allocation with excessive size value";
- the GROUND TRUTH of the individuals to evolve, namely whether the grammar forces the presence of a vulnerability or not;
- the OBJECTIVE of the evolution, that is to maximise or to minimise the fitness.

For a set of combinations of these choices, DSGE is run for 12 generations, with a population of 200 individuals and elitism of 4 individuals, with crossover and mutation occurring with probability equal to 0.9 and 0.1, respectively.

In this way, Java methods which maximise or minimise the presence of a concept in terms of signal inside the network can be found, and the evolved instances do not belong to the dataset used during the training phase. This is significant, since it enables the analysis of how the network classifies instances when the presence of a concept

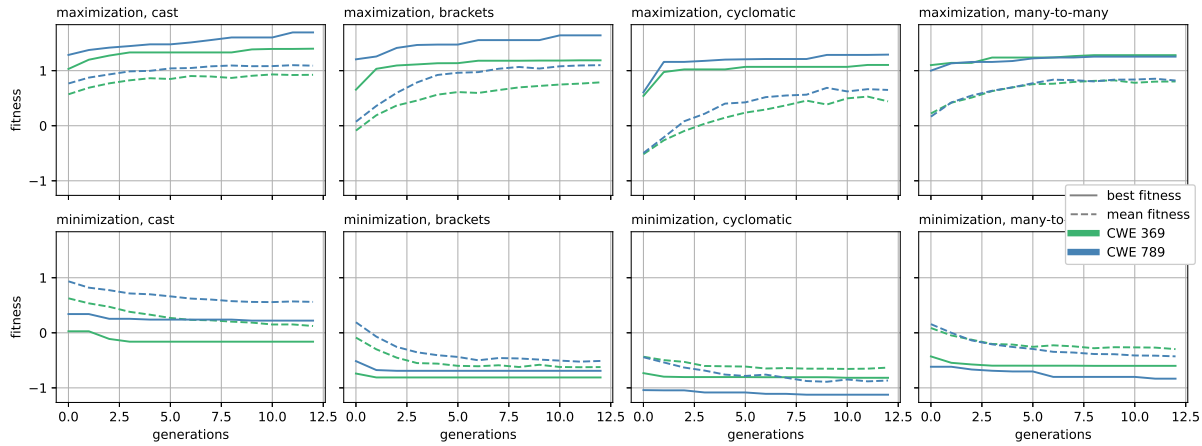


FIGURE 34: *Fitness obtained when generating input instances with DSGE, maximising or minimising the considered concepts. Notice that, in the minimization experiments, the grammar is modified in order to include a vulnerability, while the ground truth of the individual evolved when maximising the fitness is assumed to be negative.*

is enhanced or faded, and how such classification changes when the ground truth is imposed to be positive or negative.

8.3.5 *Measuring sensitivity to sub-concepts*

Finally, two analysis have been performed to study how much the prediction of the model is influenced by the concepts. The first one, which is derived from the TCAV approach [71], basically measures the percentage of instances that represent a concept, among a set of programs that are positive with respect to a classification task. Formally, for each concept c, a sample X_v of 100 methods from the dataset is considered, in which the instances are both labelled and predicted to be vulnerable by the model. Notice that only the vulnerable class has been investigated, but the same considerations can be done on the other class of safe programs. Then, the sensitivity $S_{c,v}$ has been measured by computing, $\forall x \in X_v$, the distance $d_{c,l}(f_l^*(x))$, where l is the layer referred to the best support vector $V_{c,l}$ trained on the concept c, and by computing the ratio between the number of times $d_{c,l}(f_l^*(x))$ has a positive value, and the cardinality of the sample X_v:

$$S_{c,v} = \frac{|\{x \in X_v : d_{c,l}(f_l^*(x)) > 0\}|}{|X_v|} \tag{7}$$

A second analysis is done on the instances generated via evolutionary search by measuring how accurately the network classifies the instances where one of the 4 concepts and the CWE are fixed. Two configurations have been tested:

- instances that are not vulnerable, and whose fitness value is maximised (the presence of the chosen concept is strong),
- instances that are vulnerable, and whose fitness value is minimised (the presence of the chosen concept is weak).

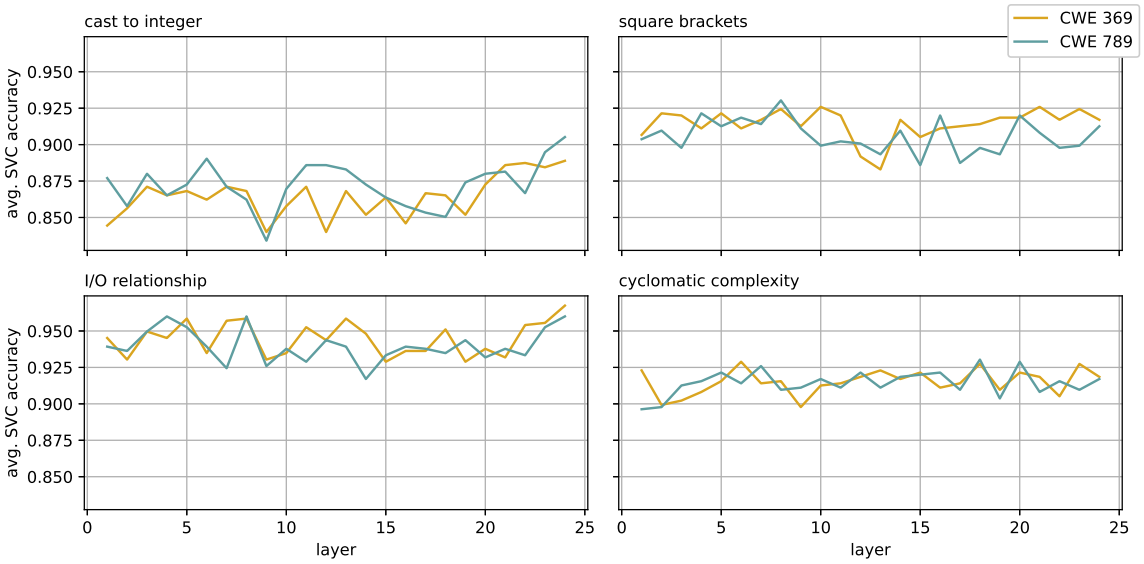


FIGURE 35: *Average accuracies obtained over 10 runs by the linear SVCs trained on different concepts.*

These configuration choices aim at checking the impact of a strong concept when the network should classify the instance as safe, and conversely in the case of instances known to be vulnerable. It is important to remind that among the four treated concepts, two are commonly considered relevant when manually looking for the examined CWE-369 or CWE-789 vulnerabilities; while two are higher level concepts, that can appear in the source code, but that are not directly related with the classification task.

This leads to 16 different experiments, and from each a pool of source code methods is eventually obtained, that is evolved according to the chosen combination of CWE, concept, and ground truth. For each experiment the network is asked to classify all the individuals generated during the 12 generations of the run, taken without repetitions (in the probable chance of duplicated individuals). Such classification is performed on two subsets of the total population of an experiment: the half population with the individuals of highest fitness values, and the half with the lowest fitness values. Finally, the relevance of a sub-concept is measured by looking at the absolute accuracy of the classification on each of the two subsets, and by analysing the relative accuracy between them. This allows to study how a concept impacts on classification, both on safe and on vulnerable source code.

8.4 RESULTS

The results of the first experimental phase, in which the emergence of the concepts is investigated, are reported in Figures 35 and 36. For each concept and for 10 runs, a linear SVC has been trained in separating the activations yielded by all the 24 layers on balanced samples of 2000 functions, where 1000 represent the concept and the other 1000 do not represent it. It can be observed that the two models

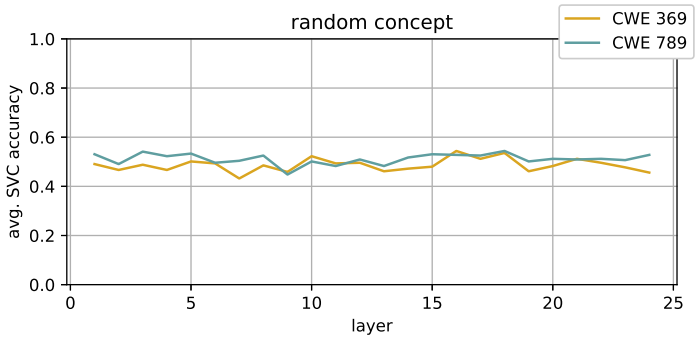


FIGURE 36: *Average accuracies obtained over 10 runs by the linear SVCs trained on random concepts.*

TABLE 11: *Sensitivity of the models to different concepts.*

Concept	**CWE-789**	**CWE-369**
Cast to integer	1.0	0.95
Square brackets	0.74	0.84
Cyclomatic complexity	0.58	0.82
Many-to-many	0.26	0.15
Random	0.49	0.51

exhibit a similar pattern for all the considered concepts, but that different concepts define problems of different difficulties: for instance, the average accuracy for the cyclomatic complexity concept is around 91% in all the layers, while for the cast to integer we move from 80% to 90%. Also, some sub-concepts (i.e. cyclomatic complexity) have a constant average accuracy along the layers, while others (e.g. cast to integer) present some increasing or decreasing trends. Note also that, as expected, the accuracy of the SVCs trained on random concepts (Figure 36) is always around 50%, thus confirming the validity of the results obtained on the well-defined concepts.

After having observed the *absolute* emergence of the concepts, their impact is also measured with respect to the original task. Two analysis have been performed: one is made by considering only vulnerable instances, and it is directly derived from the original TCAV [71], while the other is based on the classification of instances that are generated by the DSGE evolutionary algorithm.

The results of the first one are reported in Table 11, and they give interesting insights on how different concepts have a different influence on the decision. For instance, it is clear that the presence of a cast to integer has a much stronger positive effect than the many-to-many I/O relationship. Also notice that the influence of the randomly-defined concept is irrelevant, since for both the models is about 50%.

For the second sensitivity measure, 16 experiments have been performed, each corresponding to a combination of choices for concept, CWE, and maximisation or minimization of fitness, respectively on safe and on vulnerable code. As a result, each experiment gener-

ated an overall pool of around 2200 unique individuals, whose fitness ranges were polarized towards positive or negative values by the chosen optimization. Nonetheless, apart from experiments aiming at minimising for CWE-789 the presence of "cast" elements in the input, where no individual with negative fitness value was generated, in every experimental configuration, positive (when maximising) and negative (when minimising) fitness values have been easily reached after few generations (Figure 34). Recall that having fitness values spanning both positive and negative values means that each experiment generated both individuals with strong presence of a concept together with individuals with little presence of the same concept.

The accuracy of the fine-tuned transformers on each pool of individuals is presented in Table 12. Each cell corresponds to a given choice for concept, CWE, and ground truth of individuals (maximising the concept for safe instances, i.e. with ground truth equals to 0, and minimising it for vulnerable ones, i.e. with ground truth equals to 1). In each cell, the upper part reports the classifier accuracy on instances with strong presence of the concept, and conversely the lower part shows the accuracy on those with low concept signal. The numbers show quantitatively how input with more, or less, signal for a concept (i.e. the value of the decision function of the corresponding SVC, on activations induced by the instance) affects the accuracy of the classification, for safe and for vulnerable inputs. As it can be seen, the two networks have a different behavior on the evolved instances, and also with respect to different concepts.

For instance, the classifier for CWE-789, always has a very high accuracy, and it seems less impacted by the presence or the absence of the concepts. On the other hand, the transformer for CWE-369 has in general a worse accuracy. It is although interesting to analyse its accuracy values, such as they suggest different behaviours of the network when it deals with different concepts and with instances belonging to different pools. Such insights, along with the overall results are discussed in the next section.

8.5 DISCUSSION AND FINAL REMARKS

The previous sections introduced the goals, the approach, and the raw outcomes of the experiments, whose quantitative results allow the behavior of the neural classifiers to be analysed. We can discuss the outcomes of evolutionary exploration of classifiers' behavior, whose outcomes are presented in Table 12. The experiments first required a manual identification of understandable concepts that could affect the classification, and then those concepts allowed to train and select a corresponding SVC on a whole neural layer. Finally, the best SVC provided a fitness function to control the evolutionary search of input instances, either strongly or weakly related to the concept.

Overall, each row in the Table 12 gives information about the performance of a classifier on the positive and negative generated sets. In our case, the classifier trained for CWE-789 performed much better

TABLE 12: *Sensitivity of the models to different strength of concepts. Each cell corresponds to an experimental configuration, and shows the classification accuracy on instances with strong presence of the concept (top triangle) or little presence of it (bottom triangle).*

		Cast	**Brackets**	**Compl.**	**I/O**
CWE-789	vuln.	1.0 / 1.0	1.0 / 1.0	1.0 / 1.0	1.0 / 1.0
	safe	1.0 / 0.8	0.9 / 0.8	0.9 / 0.8	0.5 / 0.7
CWE-369	vuln.	0.2 / ~0	0.2 / ~0	0.3 / 0.3	0.3 / 0.1
	safe	0.3 / 0.6	0.4 / 0.6	0.7 / 0.7	0.2 / 0.5

than the one for CWE-369. Moreover, the latter shows better performance on safe instances with little presence of the sub-concepts (bottom half-cells of the bottom row in the table). These numbers also are evidence that obtained ample sets of adversarial instances have been obtained, and suggest which human expressible features in source code are more effective in deceiving the classifiers.

From data in the table, we can also move to finer observations. For instance, the CWE-369 classifier's accuracy seems immune to effects from the presence of high cyclomatic complexity in the input instances. Instead, for all the other concepts it appears that their presence increases the probability for the classifier to consider instances as vulnerable, since for instances known to be vulnerable the accuracy increases with more signal, and conversely it decreases for safe input instances, with wider or smaller variations for different concepts. The only observed anomaly for the performance of the CWE-789 classifier can be extrapolated from the worst accuracy in the second row of the table. It appears that when input methods have an array among their arguments and return an array, i.e. the "I/O" concept is strong, accuracy drops to 0.5, and this would deserve further investigation.

Finally, it is important to remark that the relevance of the results is strongly dependant on the choice of human concepts. Such a choice has to be made manually and with a good knowledge of the application domain, security of source code in this case. But sometimes, given the complex statistical basis behind the internal reasoning of neural networks, even probing them with instances associated to concepts which seemingly are unrelated to the domain could uncover unexpected mistakes in their classifying behavior. This work therefore deserves to be extended by considering different source code analysis tasks and, correspondingly, different concepts.

Part III

CLOSING REMARKS

9 CONCLUSIONS

This is the closing chapter of the thesis. Here we first resume the carried out work, and the outcomes that derived from it. Then, the final section presents some ongoing work, and discusses possible directions to further extend the evolved research lines.

9.1 OUTCOMES SUMMARY

This thesis presented an ensemble of researches whose common underlying thread was to extract knowledge from the source code with the aid of artificial intelligence techniques, both by directly operating on it, and even by analysing some of the internal dynamics that occur in the deep layers of artificial neural networks. The conducted studies, produced the following outcomes:

- a source code embedding (i.e. a vector representation) based on the abstract syntax tree (AST) able to capture both syntactical and semantic features;
- an information theoretical analysis of the behavior of internal neurons in an artificial neural system, for ranking the neurons according to their ability in solving different tasks;
- an evolutionary technique based on grammatical evolution (GE) for automatically generating "adversarial" input instances able to deceive a network trained in the detection of software vulnerabilities;
- the study, by means of dynamic structured grammatical evolution (DSGE), of the concepts that emerge in the internal layers of the CuBERT transformer and of the features that mostly affect its classification decision.

The following paragraphs discusses more in detail the outlined results.

9.1.1 *AST-based source code embedding*

The proposed source code embedding takes its inspiration from the word2vec model for natural language processing, in which the word vectors are built by means of a neural network trained in the reconstruction of the context (i.e. the neighborhood) of a word in a sentence. Due to the formal structure of programming languages, that leads to dependencies among parts of a program that can be very far from one another, in this approach words are derived from the AST nodes, and the neighborhood of a word is defined by considering paths on the

AST. Thereby, the resulting code vectors comprise both semantic (the identifiers in the AST leaves) and syntactical (the internal nodes of the AST) information. This source code embedding has been tested as input for standard classifiers trained in the classification of software vulnerabilities, achieving insightful results.

9.1.2 *Analysis of single internal neurons*

The aim was to study the ability of a deep neural network in building internal representations for specific structures or features. Simple autoencoders have been trained in the reconstruction of different kind of code vectors, and then we tested the ability of each internal neuron in classifying programs according to different binary problems (e.g. the functionality or some structural property), showing that different neurons are actually able to recognize properties that have not been specifically seen during the training phase. Furthermore, the importance of single neurons in the network has been tested from an information theoretic standpoint, showing that the defined entropy-based scoring measure is able to discriminate between important and unnecessary neurons, and even that programs that mostly stimulate a given neuron can be characterized, making this approach particularly suitable for many applications and further research.

9.1.3 *Evolving adversarial input instances*

Here, the goal was to use an evolutionary approach for synthesizing programs using, as the fitness function, the activation value of a neuron in a neural network for source code processing. A GE algorithm (i.e. a genetic algorithm able to evolve programs satisfying a given formal grammar) has been applied. The effectiveness of this approach has been assessed by showing a possible application consisting in the production of adversarial examples able to deceive a network trained in the detection of software vulnerabilities.

9.1.4 *Evolving along* concept *directions*

This work is developed along two experimental phases: the first one studies the activations of the neurons of a transformer trained in the detection of software vulnerabilities so as to identify if some human understandable concepts emerge in the network, while in the second one, program instances are generated by applying a DSGE algorithm with a fitness function that favours individuals which stimulate (or weaken) the activations in neurons where given concepts majorly emerge. Finally, the work is closed with a study on how the evolutionary pressure along the direction of a concept affects the prediction.

9.1.5 *Salient features*

This work proposes an evolutionary approach for probing the behaviour of a deep neural source code classifier by generating instances that sample its input space. By means of customized DSGE mutation operators, and by following different policies, input instances are evolved, so as to identify the features that most contribute to a particular prediction. The results showed that this approach can be effectively used for several tasks in the scope of the interpretable machine learning, such as for producing adversarial examples or for identifying blind spots or misconceptions, or in general the syntactical features that mostly affect the prediction on the original task.

9.2 FURTHER RESEARCH DIRECTIONS

Possible direct extensions of the studies that comprise this thesis have been already presented and discusses in the corresponding chapters. This last section discusses some of the insights emerged during the work conducted during the PhD and that can suggest possible broader research directions. In particular a new way to integrate AI knowledge in software development systems is discussed in Section 9.2.1, while an ongoing work on source code summarization, along with some ideas to improve the research in that context, is presented in Section 9.2.2.

9.2.1 *"Augmented" programming*

The work taken in this thesis confirmed that neural models, despite some limitations and points in which they need to be improved, are becoming effective in solving multiple tasks in the source code processing domain. On the other hand, the well known pair programming practice [145] has the key beneficial possibility of discussing with a partner (by means of alternate telling and listening quick rounds) about what is being written, since this helps the cognitive effort of writing neat and comprehensible code.

On these premise, a practical application of the research described in this thesis should be to explore the possibility of designing a revised version of the pair-programming with humans and AIs working together, by embedding in the programming platform the output of a neural network, fed with the code currently typed in by the developer. This is motivated by the fact that AI systems can output recommendations or other probable hypothesis regarding code which can be helpful to alleviate the cognitive effort needed in the development process. This idea of using AI to *support* humans, instead of *replacing* them has been already discussed in the past. For instance, in the field of chess [37, 68], teams composed by average human players and state-of-the-art chess engines (e.g. Stockfish[1]) can win against

1 https://stockfishchess.org/

both teams composed by only human experts (i.e. Grand Masters) and by an AI playing alone.

9.2.2 *Source code summarization*

Research in source code summarization, that is the description of the functionality of a program with short sentences expressed in natural language, is a topic of great interest in the software engineering community, since it can help in automatically generating software documentation, and in general can ease the effort of the developers in understanding the code they are working on. In particular, research in this direction gained popularity together with (and, probably, due to) the success that neural transformers are having in the NLP domain. In this setting, we are currently undertaking a work to point out the high sensitivity that transformers for source code have to the *natural* elements present in the source code (i.e. comments and identifiers) and the related drop in performance when such elements are ablated or masked. To this end, we developed a novel source code summarization approach based on the aid of an intermediate pseudo-language, through which it is possible to fine-tune on a source code task a state of the art summarizer for natural language (namely the BRIO model [90]), through which preliminary experiments achieved results comparable to that obtained by the state-of-the-art source code competitors (e.g. PLBART [2] and CodeBERT [48]).

The main purpose was to investigate the extent of how such naturalness affects the quality of the summaries, and the results confirmed the high sensitivity to them. This suggests many points for developing future research directions. Primarily, a general change in the approach is presumably needed: models that derive from the NLP domain seem to be effective, but the high scores that they achieve require an underlying quality of the comments and the choice of meaningful identifiers. For this reason, a re-design of these state-of-the-art approaches deserves to be explored, in order to make them less dependant from natural elements. Also, the metrics used to evaluate the quality of the summaries (e.g. BLEU [110] and ROUGE [85]) suffer from limitations, since they basically measure the overlapping among terms or n-grams, giving to each term the same importance and without considering synonyms. For this reason also a reassessments of the existing methods in lights of new metrics based on semantic similarity [144] that has been recently proposed needs to be investigated.

Finally, wondering what kind of summary should actually help a programmer or whoever has to deal with source code could be an intriguing research topic. Generally source code summarization models operate at the level of functions, that is, the resulting summary provides a description in natural language of the functionality of the function itself. Possible works in this direction should be to investigate if this kind of outcome is really supportive for programmers, and to eventually design other summarization paradigms. For instance, the common practice of program slicing [143], that allows to

isolate the program statements that may affect the values of a given set of variables at some point of interest, can be combined with summarization approaches to produce high level descriptions of *slices* of programs that are referred only to the variables of interest.

BIBLIOGRAPHY

[1] Amina Adadi and Mohammed Berrada. «Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)». In: *IEEE Access* 6 (2018), pp. 52138–52160.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. «Unified Pre-training for Program Understanding and Generation». In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, June 2021, pp. 2655–2668.

[3] Miltiadis Allamanis. «The adverse effects of code duplication in machine learning models of code». In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2019, pp. 143–153.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. «A Survey of Machine Learning for Big Code and Naturalness». In: *ACM Comput. Surv.* 51.4 (2018), 81:1–81:37.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. «Learning to Represent Programs with Graphs». In: *Proceedings of 6th International Conference on Learning Representations, ICLR*. 2018.

[6] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. «A Convolutional Attention Network for Extreme Summarization of Source Code». In: *Proceedings of the 33nd International Conference on Machine Learning, ICML*. 2016, pp. 2091–2100.

[7] Frances E. Allen. «Control Flow Analysis». In: *SIGPLAN Not.* 5.7 (1970), pp. 1–19.

[8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. «code2seq: Generating Sequences from Structured Representations of Code». In: *7th International Conference on Learning Representations, ICLR*. OpenReview.net, 2019.

[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. «A general path-based representation for predicting program properties». In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 2018, pp. 404–419.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. «code2vec: learning distributed representations of code». In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 40:1–40:29.

[11] João Antunes, Nuno Ferreira Neves, and Paulo Jorge Veríssimo. «Detection and Prediction of Resource-Exhaustion Vulnerabilities». In: *19th International Symposium on Software Reliability Engineering (ISSRE)*. 2008, pp. 87–96.

[12] Shushan Arakelyan, Christophe Hauser, Erik Kline, and Aram Galstyan. *Towards Learning Representations of Binary Executable Files for Security Tasks*. arXiv. 2020. eprint: 2002.03388 (cs.CR). URL: https://arxiv.org/abs/2002.03388.

[13] Thomas Bäck and Hans-Paul Schwefel. «An Overview of Evolutionary Algorithms for Parameter Optimization». In: *Evolutionary Computation* 1.1 (1993), pp. 1–23.

[14] John W. Backus et al. «Revised report on the algorithmic language ALGOL 60». In: *Comput. J.* 5.4 (1963), pp. 349–367.

[15] Francesco Barchi, Emanuele Parisi, Gianvito Urgese, Elisa Ficarra, and Andrea Acquaviva. «Exploration of convolutional neural network models for source code classification». In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104075.

[16] Anthony Bau et al. «Identifying and Controlling Important Neurons in Neural Machine Translation». In: *7th International Conference on Learning Representations, ICLR*. OpenReview.net, 2019.

[17] Paul E. Black. *Juliet 1.3 Test Suite: Changes From 1.2*. Technical Note. NIST National Institute for Standard and Technology, 2018.

[18] Tim Boland and Paul E. Black. «The Juliet C/C++ and Java Test Suite». In: *Computer (IEEE Computer)* 45 (2012).

[19] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. «A Training Algorithm for Optimal Margin Classifiers». In: *5th Annual ACM Conference on Computational Learning Theory, COLT*. ACM, 1992, pp. 144–152.

[20] Leo Breiman. «Random Forests». In: *Mach. Learn.* 45.1 (2001), pp. 5–32.

[21] Ruven E. Brooks. «Towards a Theory of the Comprehension of Computer Programs». In: *Int. J. Man Mach. Stud.* 18.6 (1983), pp. 543–554.

[22] Marcel Bruch, Martin Monperrus, and Mira Mezini. «Learning from examples to improve code completion systems». In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ed. by Hans van Vliet and Valérie Issarny. ACM, 2009, pp. 213–222.

[23] David Brumley, Dawn Xiaodong Song, Tzi-cker Chiueh, Rob Johnson, and Huijia Lin. «RICH: Automatically Protecting Against Integer-Based Vulnerabilities». In: *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2007.

[24] S. N. Cant, D. Ross Jeffery, and Brian Henderson-Sellers. «A conceptual model of cognitive complexity of elements of the programming process». In: *Inf. Softw. Technol.* 37.7 (1995), pp. 351–362.

[25] Tom Castle and Colin G. Johnson. «Positional Effect of Crossover and Mutation in Grammatical Evolution». In: *Proceedings of 13th European Conference Genetic Programming EuroGP*. Vol. 6021. Lecture Notes in Computer Science. Springer, 2010, pp. 26–37.

[26] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. «Deep Learning Based Vulnerability Detection: Are We There Yet?» In: *IEEE Trans. Software Eng.* 48.9 (2022), pp. 3280–3296.

[27] Dhivya Chandrasekaran and Vijay Mago. «Evolution of Semantic Similarity - A Survey». In: *ACM Comput. Surv.* 54.2 (2022), 41:1–41:37.

[28] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. arXiv. 2021. eprint: 2107.03374 (cs.LG). URL: https://arxiv.org/abs/2107.03374.

[29] François Chollet et al. *Keras*. https://keras.io. 2015.

[30] Noam Chomsky. «Three models for the description of language». In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[31] Crispin Cowan et al. «FormatGuard: Automatic Protection From printf Format String Vulnerabilities». In: *USENIX Security Symposium*. Vol. 91. Washington, DC. 2001.

[32] Matej Crepinsek, Shih-Hsi Liu, and Marjan Mernik. «Exploration and exploitation in evolutionary algorithms: A survey». In: *ACM Comput. Surv.* 45.3 (2013), 35:1–35:33.

[33] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. «Utilizing source code embeddings to identify correct patches». In: *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE. 2020, pp. 18–25.

[34] Fahim Dalvi et al. «What Is One Grain of Sand in the Desert? Analyzing Individual Neurons in Deep NLP Models». In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 6309–6317.

[35] Cristina David and Daniel Kroening. «Program synthesis: challenges and opportunities». In: *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences* 375 (Oct. 2017).

[36] Jesse Davis and Mark Goadrich. «The relationship between Precision-Recall and ROC curves». In: *Machine Learning, Proceedings of the 23rd International Conference (ICML)*. Vol. 148. ACM International Conference Proceeding Series. ACM, 2006, pp. 233–240.

[37] David De Cremer and Garry Kasparov. «AI should augment human intelligence, not replace it». In: *Harvard Business Review* 18 (2021).

[38] Alvaro Fernández Del Carpio and Leonardo Bermón Angarita. «Trends in Software Engineering Processes using Deep Learning: A Systematic Literature Review». In: *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 445–454.

[39] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[40] Will Dietz, Peng Li, John Regehr, and Vikram Adve. «Understanding integer overflow in C/C++». In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.1 (2015), pp. 1–29.

[41] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. «Feature location in source code: a taxonomy and survey». In: *Journal of software: Evolution and Process* 25.1 (2013), pp. 53–95.

[42] Yinpeng Dong, Fan Bao, Hang Su, and Jun Zhu. *Towards Interpretable Deep Neural Networks by Leveraging Adversarial Examples*. arXiv. 2019. eprint: 1901.09035 (cs.LG). URL: http://arxiv.org/abs/1901.09035.

[43] Finale Doshi-Velez and Been Kim. *A Roadmap for a Rigorous Science of Interpretability*. arXiv. 2017. eprint: 1702.08608 (stat.ML). URL: http://arxiv.org/abs/1702.08608.

[44] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. «Fear the ear: discovering and mitigating execution after redirect vulnerabilities». In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 251–262.

[45] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. «Visualizing higher-layer features of a deep network». In: *University of Montreal* 1341.3 (2009), p. 1.

[46] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. «Why does unsupervised pre-training help deep learning?» In: *Proceedings of the 13th international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 201–208.

[47] Y. Fang, S. Han, C. Huang, and R. Wu. «TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology». In: *PLoS ONE* 14(11) (2019).

[48] Zhangyin Feng et al. «CodeBERT: A Pre-Trained Model for Programming and Natural Languages». In: *Findings of the Association for Computational Linguistics: EMNLP*. Vol. EMNLP 2020. Findings of ACL. Association for Computational Linguistics, 2020, pp. 1536–1547.

[49] Michael Fenton et al. «PonyGE2: grammatical evolution in Python». In: *Companion Material Proceedings of Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1194–1201.

[50] Claudio Ferretti and Martina Saletta. «Deceiving neural source code classifiers: finding adversarial examples with grammatical evolution». In: *GECCO '21: Genetic and Evolutionary Computation Conference, Companion Volume*. ACM, 2021, pp. 1889–1897.

[51] Claudio Ferretti and Martina Saletta. «Do Neural Transformers Learn Human-Defined Concepts? An Extensive Study in Source Code Processing Domain». In: *Algorithms* 15.12 (2022).

[52] Shlok Gilda. «Source code classification using Neural Networks». In: *2017 14th international joint conference on computer science and software engineering (JCSSE)*. IEEE. 2017, pp. 1–6.

[53] Leilani H Gilpin et al. «Explaining explanations: An overview of interpretability of machine learning». In: *IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE. 2018, pp. 80–89.

[54] Jacob Goldberger, Sam T. Roweis, Geoffrey E. Hinton, and Ruslan Salakhutdinov. «Neighbourhood Components Analysis». In: *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS*. 2004, pp. 513–520.

[55] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[56] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. «Explaining and Harnessing Adversarial Examples». In: *3rd International Conference on Learning Representations, ICLR*. 2015.

[57] Jeremiah Grossman, Seth Fogie, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS attacks: cross site scripting exploits and defense*. Syngress, 2007.

[58] David Gunning et al. «XAI—Explainable artificial intelligence». In: *Science robotics* 4.37 (2019).

[59] William G. Halfond, Jeremy Viegas, and Alessandro Orso. «A classification of SQL-injection attacks and countermeasures». In: *Proceedings of the IEEE international symposium on secure software engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[60] Warren He, Bo Li, and Dawn Song. «Decision Boundary Analysis of Adversarial Examples». In: *6th International Conference on Learning Representations, ICLR*. OpenReview.net, 2018.

[61] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. «On domain knowledge and novelty to improve program synthesis performance with grammatical evolution». In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO*. Ed. by Anne Auger and Thomas Stützle. ACM, 2019, pp. 1039–1046.

[62] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[63] Nwokedi Idika and Aditya P. Mathur. «A survey of malware detection techniques». In: *Purdue University* 48 (2007), pp. 2007–2.

[64] Licheng Jiao and Jin Zhao. «A Survey on the New Generation of Deep Learning in Image Processing». In: *IEEE Access* 7 (2019), pp. 172231–172263.

[65] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. «Learning and evaluating contextual embedding of source code». In: *Proceedings of the 37th International Conference on Machine Learning, ICML*. Proceedings of Machine Learning Research. PMLR, 2020.

[66] Asif Karim, Sami Azam, Bharanidharan Shanmugam, Krishnan Kannoorpatti, and Mamoun Alazab. «A Comprehensive Survey for Intelligent Spam Email Detection». In: *IEEE Access* 7 (2019), pp. 168261–168295.

[67] Hamid Karimi, Tyler Derr, and Jiliang Tang. *Characterizing the Decision Boundary of Deep Neural Networks*. arXiv. 2019. eprint: 1912.11460 (cs.LG). URL: http://arxiv.org/abs/1912.11460.

[68] Garry Kasparov. «The chess master and the computer». In: *The New York Review of Books* 57.2 (2010), pp. 16–19.

[69] Krishna M. Kavi, Bill P. Buckles, and U. Narayan Bhat. «A formal definition of data flow graph models». In: *IEEE Transactions on computers* 35.11 (1986), pp. 940–948.

[70] Salman Khan et al. «Transformers in Vision: A Survey». In: *ACM Comput. Surv.* 54.10s (2022).

[71] Been Kim et al. «Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)». In: *35th International Conference on Machine Learning, ICML*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 2673–2682.

[72] Yoon Kim. «Convolutional Neural Networks for Sentence Classification». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP*. ACL, 2014, pp. 1746–1751.

[73] Donald E. Knuth. «Semantics of Context-Free Languages». In: *Math. Syst. Theory* 2.2 (1968), pp. 127–145.

[74] Xianglong Kong, Supeng Kong, Ming Yu, and Chengjie Du. «Joint Embedding of Semantic and Statistical Features for Effective Code Search». In: *Applied Sciences* 12.19 (2022), p. 10002.

[75] Sotiris B. Kotsiantis. «Supervised Machine Learning: A Review of Classification Techniques». In: *Emerging Artificial Intelligence Applications in Computer Engineering - Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Vol. 160. Frontiers in Artificial Intelligence and Applications. IOS Press, 2007, pp. 3–24.

[76] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, 1992.

[77] Pat Langley. «The changing science of machine learning». In: *Mach. Learn.* 82.3 (2011), pp. 275–279.

[78] Quoc V. Le and Tomás Mikolov. «Distributed Representations of Sentences and Documents». In: *Proceedings of the 31th International Conference on Machine Learning, ICML*. 2014, pp. 1188–1196.

[79] Quoc V. Le et al. «Building high-level features using large scale unsupervised learning». In: *Proceedings of the 29th International Conference on Machine Learning, ICML*. PMLR, 2012, pp. 507–514.

[80] Triet Huynh Minh Le, Hao Chen, and Muhammad Ali Babar. «Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges». In: *ACM Comput. Surv.* 53.3 (2021), 62:1–62:38.

[81] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. «Code Completion with Neural Attention and Pointer Networks». In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 4159–4165.

[82] Zhen Li et al. «VulDeePecker: A Deep Learning-Based System for Vulnerability Detection». In: *25th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2018.

[83] Zhen Li et al. «A Comparative Study of Deep Learning-Based Vulnerability Detection System». In: *IEEE Access* 7 (2019), pp. 103184–103197.

[84] Zhen Li et al. «SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities». In: *IEEE Trans. Dependable Secur. Comput.* 19.4 (2022), pp. 2244–2258.

[85] Chin-Yew Lin. «ROUGE: A Package for Automatic Evaluation of Summaries». In: *Text Summarization Branches Out*. Association for Computational Linguistics, 2004, pp. 74–81.

[86] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. «Software Vulnerability Detection Using Deep Neural Networks: A Survey». In: *Proc. IEEE* 108.10 (2020), pp. 1825–1848.

[87] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. «A survey of transformers». In: *AI Open* (2022).

[88] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. «Software vulnerability discovery techniques: A survey». In: *2012 fourth international conference on multimedia information networking and security*. IEEE. 2012, pp. 152–156.

[89] Fang Liu et al. «A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning». In: *Proceedings of the 28th International Conference on Program Comprehension, ICPC*. ACM, 2020, pp. 37–47.

[90] Yixin Liu, Pengfei Liu, Dragomir R. Radev, and Graham Neubig. «BRIO: Bringing Order to Abstractive Summarization». In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*. Association for Computational Linguistics, 2022, pp. 2890–2903.

[91] Nuno Lourenço, Filipe Assunção, Francisco B Pereira, Ernesto Costa, and Penousal Machado. «Structured grammatical evolution: a dynamic approach». In: *Handbook of Grammatical Evolution*. Springer, 2018, pp. 137–161.

[92] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. «Unveiling the properties of structured grammatical evolution». In: *Genetic Programming and Evolvable Machines* 17.3 (2016), pp. 251–289.

[93] Nuno Lourenço, Francisco B. Pereira, and Ernesto Costa. «SGE: A Structured Representation for Grammatical Evolution». In: *12th International Conference of Artificial Evolution - Evolution Artificielle, EA*. Vol. 9554. Lecture Notes in Computer Science. Springer, 2015, pp. 136–148.

[94] Sean Luke. *Essentials of Metaheuristics*. second. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/. Lulu, 2013.

[95] Henry B. Mann and Donald R. Whitney. «On a test of whether one of two random variables is stochastically larger than the other». In: *The annals of mathematical statistics* (1947), pp. 50–60.

[96] Anneliese von Mayrhauser and A. Marie Vans. «Program Comprehension During Software Maintenance and Evolution». In: *Computer* 28.8 (1995), pp. 44–55.

[97] Thomas J. McCabe. «A Complexity Measure». In: *IEEE Trans. Softw. Eng.* 2.4 (1976), pp. 308–320.

[98] Thomas McGrath et al. «Acquisition of chess knowledge in alphazero». In: *Proceedings of the National Academy of Sciences* 119.47 (2022), e2206625119.

[99] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. «Sentiment analysis algorithms and applications: A survey». In: *Ain Shams engineering journal* 5.4 (2014), pp. 1093–1113.

[100] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. «Distributed Representations of Words and Phrases and their Compositionality». In: *Advances in Neural Information Processing Systems 26, Proceedings of NIPS*. 2013, pp. 3111–3119.

[101] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022. URL: https://christophm.github.io/interpretable-ml-book.

[102] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. «Convolutional Neural Networks over Tree Structures for Programming Language Processing». In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 1287–1293.

[103] Anh Nguyen, Jason Yosinski, and Jeff Clune. «Deep neural networks are easily fooled: High confidence predictions for unrecognizable images». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 427–436.

[104] Anh Mai Nguyen, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. «Synthesizing the preferred inputs for neurons in neural networks via deep generator networks». In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*. 2016, pp. 3387–3395.

[105] Michael O'Neill, Miguel Nicolau, and Alexandros Agapitos. «Experiments in program synthesis with grammatical evolution: A focus on Integer Sorting». In: *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*. IEEE, 2014, pp. 1504–1511.

[106] Michael O'Neill and Conor Ryan. «Grammatical evolution». In: *IEEE Trans. Evol. Comput.* 5.4 (2001), pp. 349–358.

[107] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. «Feature visualization». In: *Distill* 2.11 (2017), e7.

[108] Aleph One. «Smashing the stack for fun and profit». In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[109] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. «SoK: Security and Privacy in Machine Learning». In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 399–414.

[110] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. «Bleu: a Method for Automatic Evaluation of Machine Translation». In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL, 2002, pp. 311–318.

[111] Erik Pitzer and Michael Affenzeller. «A Comprehensive Survey on Fitness Landscape Analysis». In: *Recent Advances in Intelligent Engineering Systems*. Vol. 378. Studies in Computational Intelligence. Springer, 2012, pp. 161–191.

[112] Michael Pradel and Koushik Sen. «DeepBugs: a learning approach to name-based bug detection». In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 147:1–147:25.

[113] Erwin Quiring, Alwin Maier, and Konrad Rieck. «Misleading Authorship Attribution of Source Code using Adversarial Learning». In: *28th USENIX Security Symposium*. USENIX Association, 2019, pp. 479–496.

[114] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. *Learning to Generate Reviews and Discovering Sentiment*. arXiv. 2017. eprint: 1704.01444 (cs.LG). URL: https://arxiv.org/pdf/1704.01444.pdf.

[115] Gabrielle Ras, Ning Xie, Marcel van Gerven, and Derek Doran. «Explainable Deep Learning: A Field Guide for the Uninitiated». In: *J. Artif. Intell. Res.* 73 (2022), pp. 329–396.

[116] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. «Probabilistic model for code with decision trees». In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 2016, pp. 731–747.

[117] Radim Řehůřek and Petr Sojka. «Software Framework for Topic Modelling with Large Corpora». In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[118] Frank Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain». In: *Psychological Review* (1958), pp. 65–386.

[119] Franz Rothlauf and Marie Oetzel. «On the Locality of Grammatical Evolution». In: *Proceedings of 9th European Conference on Genetic Programming EuroGP*. Vol. 3905. Lecture Notes in Computer Science. Springer, 2006, pp. 320–330.

[120] Peter J. Rousseeuw. «Silhouettes: A graphical aid to the interpretation and validation of cluster analysis». In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65.

[121] Herbert Rubenstein and John B. Goodenough. «Contextual correlates of synonymy». In: *Commun. ACM* 8.10 (1965), pp. 627–633.

[122] Rebecca L. Russell et al. «Automated Vulnerability Detection in Source Code Using Deep Representation Learning». In: *Proceedings of 17th IEEE International Conference on Machine Learning and Applications, ICMLA*. IEEE, 2018, pp. 757–762.

[123] Conor Ryan, J. J. Collins, and Michael O'Neill. «Grammatical Evolution: Evolving Programs for an Arbitrary Language». In: *Proceedings of the First European Workshop on Genetic Programming*. Vol. 1391. LNCS. Springer-Verlag, 1998, pp. 83–96.

[124] Barbara G. Ryder. «Constructing the call graph of a program». In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 216–226.

[125] Ruslan Salakhutdinov and Geoffrey E. Hinton. «Semantic hashing». In: *International Journal of Approximate Reasoning* 50.7 (2009), pp. 969–978.

[126] Martina Saletta and Claudio Ferretti. «A Neural Embedding for Source Code: Security Analysis and CWE Lists». In: *IEEE Intl. Conf. on Dependable, Autonomic and Secure Computing, DASC/PiCom/CBDCom/CyberSciTech*. IEEE, 2020, pp. 523–530.

[127] Martina Saletta and Claudio Ferretti. *Mining Program Properties From Neural Networks Trained on Source Code Embeddings*. arXiv. 2021. eprint: 2103.05442 (cs.SE). URL: https://arxiv.org/abs/2103.05442.

[128] Martina Saletta and Claudio Ferretti. «A Grammar-based Evolutionary Approach for Assessing Deep Neural Source Code Classifiers». In: *IEEE Congress on Evolutionary Computation, CEC*. IEEE, 2022, pp. 1–8.

[129] Martina Saletta and Claudio Ferretti. «Towards the evolutionary assessment of neural transformers trained on source code». In: *GECCO '22: Genetic and Evolutionary Computation Conference, Companion Volume*. ACM, 2022, pp. 1770–1778.

[130] Martin Schrimpf et al. «Brain-score: Which artificial neural network for object recognition is most brain-like?» In: *BioRxiv* (2020), p. 407007.

[131] Claude E. Shannon. «A mathematical theory of communication». In: *Bell Syst. Tech. J.* 27.4 (1948), pp. 623–656.

[132] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. «Learning Important Features Through Propagating Activation Differences». In: *34th International Conference on Machine Learning, ICML*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 3145–3153.

[133] Janet Siegmund et al. «Understanding understanding source code with functional magnetic resonance imaging». In: *36th International Conference on Software Engineering, ICSE*. ACM, 2014, pp. 378–389.

[134] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited*. Version dated 17 May 2019. URL: https://leanpub.com/javaparservisited.

[135] Dominik Sobania and Franz Rothlauf. «Challenges of Program Synthesis with Grammatical Evolution». In: *Proceedings of Genetic Programming - 23rd European Conference (EuroGP), held as Part of EvoStar*. Vol. 12101. Lecture Notes in Computer Science. Springer, 2020, pp. 211–227.

[136] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. «IntelliCode compose: code generation using transformer». In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 1433–1443.

[137] Florian Tramèr, Nicolas Papernot, Ian J. Goodfellow, Dan Boneh, and Patrick D. McDaniel. *The Space of Transferable Adversarial Examples*. arXiv. 2017. eprint: 2103.05442 (stat.ML). URL: http://arxiv.org/abs/1704.03453.

[138] Laurens Van der Maaten and Geoffrey Hinton. «Visualizing data using t-SNE». In: *Journal of machine learning research* 9.11 (2008).

[139] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. «Neural Program Repair by Jointly Learning to Localize and Repair». In: *7th International Conference on Learning Representations, ICLR*. OpenReview.net, 2019.

[140] Ashish Vaswani et al. «Attention is All you Need». In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. 2017, pp. 5998–6008.

[141] Yao Wan et al. «Improving automatic source code summarization via deep reinforcement learning». In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 2018, pp. 397–407.

[142] Yu Wang, Yu Dong, Xuesong Lu, and Aoying Zhou. «GypSum: learning hybrid representations for code summarization». In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC*. ACM, 2022, pp. 12–23.

[143] Mark Weiser. «Program Slicing». In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357.

[144] John Wieting, Taylor Berg-Kirkpatrick, Kevin Gimpel, and Graham Neubig. «Beyond BLEU: Training Neural Machine Translation with Semantic Similarity». In: *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL*. Association for Computational Linguistics, 2019, pp. 4344–4355.

[145] Laurie Williams, Robert R Kessler, Ward Cunningham, and Ron Jeffries. «Strengthening the case for pair programming». In: *IEEE software* 17.4 (2000), pp. 19–25.

[146] Fabian Yamaguchi, Felix "FX" Lindner, and Konrad Rieck. «Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning». In: *Proceedings of 5th USENIX Workshop on Offensive Technologies, WOOT'11*. USENIX Association, 2011, pp. 118–127.

[147] Noam Yefet, Uri Alon, and Eran Yahav. «Adversarial examples for models of code». In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 162:1–162:30.

[148] Suan Hsi Yong and Susan Horwitz. «Protecting C programs from attacks via invalid pointer dereferences». In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE*. ACM, 2003, pp. 307–316.

[149] Shujian Yu and José C. Príncipe. «Understanding autoencoders with information theoretic concepts». In: *Neural Networks* 117 (2019), pp. 104–123.

[150] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

[151] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. arXiv. 2012. eprint: 1212.5701 (cs.LG). URL: https://arxiv.org/pdf/1212.5701.pdf.

[152] Matthew D. Zeiler and Rob Fergus. «Visualizing and Understanding Convolutional Networks». In: *13th European Conference of Computer Vision ECCV*. Vol. 8689. Lecture Notes in Computer Science. Springer, 2014, pp. 818–833.

[153] Chunyan Zhang et al. «A Survey of Automatic Source Code Summarization». In: *Symmetry* 14.3 (2022), p. 471.

[154] Huangzhao Zhang et al. «Generating Adversarial Examples for Holding Robustness of Source Code Processing Models». In: *The 34th AAAI Conference on Artificial Intelligence, AAAI, The 32nd Innovative Applications of Artificial Intelligence Conference, IAAI, The 10th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI*. AAAI Press, 2020, pp. 1169–1176.

[155] Jian Zhang et al. «A Novel Neural Source Code Representation Based on Abstract Syntax Tree». In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 783–794.

[156] Fuzhen Zhuang et al. «A comprehensive survey on transfer learning». In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.

[157] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. «Adversarial Attacks on Neural Networks for Graph Data». In: *24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, 2018, pp. 2847–2856.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".