SAPIENZA
UNIVERSITÀ DI ROMA

**Sapienza University of Rome**

Department of Computer, Control, and Management Engineering Antonio Ruberti
*Ph.D. in Engineering in Computer Science*

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# Application of Language Models on Code Analysis

Advisor
**Prof. Leonardo Querzoni**

Co-Advisors
**Prof. Giuseppe Antonio Di Luna**
**Prof. Riccardo Lazzeretti**

Candidate
**Francesca Console**
**1327819**

**Academic Year MMXXIII-MMXXIV (XXXVI cycle)**

*A Marco, a mamma e papà, a Johnny, a Sara. Grazie per avermi supportato durante questo bellissimo viaggio.*

# Abstract

Code analysis is a key topic for improving software quality and efficiency. This analysis becomes even more important for securing code against potential cyber-attacks.

However, manual analysis of code, especially for the binary one, is complicated and error-prone. Therefore, the investigation of new automatic techniques for code analysis is research topic of great interest. As suggested by the "naturalness hypothesis", the code exhibits similar statistical properties to natural languages. As a consequence, techniques used for natural language processing can be also applied to analyze source and binary code. For this reason, recent research applies neural language models on code analysis, achieving significant results.

In line with this research trend, the two contributions of the thesis are focused on the application of deep learning to analysis of code written in high-level and low-level programming languages.

The first contribution of the thesis introduces a benchmark designed to evaluate models for binary code representation. The tool can be used to test and compare the performance of these models on various binary function tasks.

The second contribution, on the other hand, focuses on the application of neural networks for analyzing source code. The contribution investigates the application of neural language models for detecting code smells, that represent poor design choices potentially impacting the code quality.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Recent years have seen significant advances in technology. Computations that were once performed exclusively by supercomputers are now being executed by common devices, such as smartphones and personal computers. These improvements have led to the spread of digital devices in various sectors. Health care, mobility and industrial automation are just some of the fields that have been revolutionized by these technologies. Fundamental components of the digital transformation are the embedded systems, microprocessor-based devices designed for executing specific tasks. Having limited functionality, these systems can be configured in a very specialized way using simpler technologies than general purpose systems [56]. However, high-quality code is necessary to accomplish high performance and reliability within this context.

A key practice to achieve code quality is the *code analysis*. This is a process of revision that allows developers to understand the code implementation and identify potential improvements to be made [70]. Code analysis assumes even a greater significance for enhancing code security, enabling the identification of potential vulnerabilities before their exploitation. Depending on the requirements, the analysis can be conducted on either source code or binary code. While manual review provides a deep understanding of the code logic, it is generally resource and time consuming. In addition, this type of analysis becomes mostly inapplicable on large code bases. For this reason, specific software is often used to facilitate the process of review. These traditional tools, however, still require experienced technicians to configure them. For instance, the manual definition of rules is required for identifying code smells using metrics-based approaches.

These limitations are encouraging researchers to explore more dynamic and automated methods of analysis, focused on the use of deep neural networks (DNNs). As also hinted by Allamanis et al. [3], code corpora present similar properties to natural languages. Therefore, methodologies already used and validated for natural language processing (NLP) can be also applied in the context of code analysis. For this reason, recent research trends are focused on the adaptation of language models (LMs) to code analysis. This approach achieved significant results on a variety of tasks, resulting also in the definition of specialized models on source and binary code, such as CodeBERT [41] and SAFE [87].

Building on current research, the contributions of thesis are focused on studying different aspects of source and binary code using neural language models.

In particular, we first evaluate models for binary code representation. These architectures are dedicated on solving specific task on binaries, such as similarity and code search. To improve generalization, models can be trained and tested through specific benchmarks containing multiple tasks. These are key tools for the NLP community to objectively compare the effectiveness of various models. However, the literature still lack of benchmarks for models on binary code representation. For this reason, we develop a multi-task benchmark exploring different features of binary functions.

While binary code analysis is of fundamental importance, an in-depth study of source code is also necessary. The second contribution of the thesis discusses the application of deep learning on source code analysis. In particular, we aim to evaluate the potential of DNNs for automating the detection of code smells in functions. Code smells represent weak design choices that could negatively impact the code quality. Timely detection is crucial for developers, as it allows to promptly fix code weaknesses and prevent cyber-attacks. Traditional detection techniques rely on metrics and heuristics, necessitating the ongoing supervision of specialists and a significant use of resources. To overcome these problems, we intend to develop a reliable and automated approach for code smell identification. For this reason, the second contribution of the thesis assesses detection capabilities of different large language models (LLMs) on a variety of code smells.

## 1.1 Main Contributions

In this thesis we investigate the capabilities of deep learning in various tasks, defined for both binary and source code. In the following, we give an overview of the main contributions of the thesis. The first contribution focuses on the development of a benchmark for binary function representation models. The second contribution analyzes the application of LLMs in automatically identifying code smells within functions.

### 1.1.1 Benchmark for Binary Code Representation Models

Recent research has effectively applied deep learning techniques, originally used in language models, on the binary code analysis. Leveraging the same approaches employed for natural languages, these models demonstrated remarkable abilities to understand and represent binary code features. This approach showed state-of-the-art performance in a variety of binary tasks, including the recovering of compiler provenance and evaluation of binary similarity between code snippets.

To guarantee the generalizability of these models, their performance must be tested on various aspects of the code. A common solution applied in NLP field is the evaluation of models through widely accepted benchmarks, including multiple problems to be solved on text corpora [67, 98, 123]. In the context of binary code analysis, however, there is no benchmark for testing models on multiple tasks. This absence causes two main problems. The first is the limited generalizability of the models, while the latter is the incomparability of models' results.

For these reasons, the first contribution of the thesis introduce the development of BinBench, a multi-task benchmark designed for assessing models on binary functions. The benchmark is composed of five tasks, each one requiring DNN architectures to infer specific information regarding binary functions. The benchmark also includes a large dataset, obtained from the compilation of 131 Arch Linux packages. The dataset includes 4'408'191 binary functions stored in JSON format, along with the original binary files they were extracted from. To ensure their validity, the benchmark's tasks have been tested with various deep learning models.

The work has been proposed in the paper "BinBench: a benchmark for x64 portable operating system interface binary function representations" [25], which was published in the journal PeerJ Computer Science in 2023.

### 1.1.2 Code Smell Detection

With the growth of devices connected to the internet, the number of cyber attacks is also increasing. Exploiting a single vulnerability in a well-known software can spread to multiple devices, making them vulnerable to the same attacks. To prevent these risks, the examination of code is a fundamental tool. This process involves the investigation of several aspects of the code, including the identification of code smells. However, traditional code smell detection methods requires time and resources. In addition, these techniques often suffer from subjectivity, as technicians need to manually define the heuristics and metrics used for identification. For these reasons, recent works are focused on automating code smell detection with deep learning approaches [52, 82].

Following this trend, the second contribution of the thesis analyzes the application of deep neural networks for detecting method-level code smells. Specifically, we first select a large collection of code smells, including over 4'000'000 samples. Using the resulting dataset, we fine-tune and test four different LLMs in automatically detecting various types of code smells. Furthermore, we investigate the impact of different pre-trainings on the models' performance. We compare three large language models pre-trained on code to one pre-trained on natural language, demonstrating the significant benefit of code-specific pre-training in code smell detection.

# Chapter 2

# Background on Deep Neural Networks

Deep neural networks have become popular tools for solving complex tasks in different fields, including natural language processing, image processing, and audio classification. The popularity of DNNs derives from their ability of solving complex tasks with high accuracy and adaptive approach. This is achieved by learning an internal representation of the objects under analysis. Deep learning models, in fact, are able to extract very complex features from raw data, by creating representations that are expressed in terms of other, simpler representations [49]. Due to their versatility, NLP researchers developed several architectures. Examples of these models are BERT [33], GPT-2 [57], GPT-3 [16] and GPT-4 [2], which use the learned representations to solve the most different tasks acting as *universal unsupervised task solvers* [101, 131].

Similarly to other research fields, also code analysis is successfully applying DNNs for solving widely different tasks, such as the binary similarity and the reconstruction of stripped symbols. As in fact suggested by the "naturalness hypothesis" [3], the code, even in its binary form, is a medium of communication between human and machines. Therefore, code has statistical properties comparable to natural language corpora that can be analyzed using the same machine learning techniques and statistical methods applied to natural languages. An empirical proof of the naturalness hypothesis is that many models developed for NLP were used on code analysis with impressive results. Furthermore, as mentioned earlier, the assumption holds not only for source code but also for binary code. Several researches on binary domain have, indeed, applied DNNs borrowed from NLP community, often adapting their architectures with only minor changes. As a result, DNNs showed state-of-the-art performance also in binary code analysis. Examples of this approach are the self-attentive recurrent neural network (RNN) used in SAFE [87], the word2vec [90] architecture used in Eklavya [22], and the transformer-based model used in CodeBERT, jTrans and Palmtree [41, 77, 124].

This chapter analyzes the application of various DNN architectures in the field of natural language processing. Specifically, we initially describe the role of language models within NLP. Subsequently, we provide a detailed description of different DNN architectures.

## 2.1 Language Models

The main task addressed by natural language processing is to let machines understand and manipulate information from human (i.e., "natural") languages. To achieve this goal, NLP research uses fundamental tools called *language models*. LMs are probabilistic algorithms that are able to understand natural languages [19]. This ability is used for solving a variety of tasks, such as text generation and question answering.

LMs can be implemented as pure statistical models (e.g., n-gram model) or using neural networks. Recently, researchers also developed more advanced models known as *large language models*, having outstanding performance in learning and solving various natural language processing tasks.

LMs are able to estimate the likelihood associated to the strings occurring in a natural language, such as the words in the English language. This estimation is achieved considering the set of conditional probabilities $P(w_k|w_1^{k-1})$ representing the production of strings in the language considered, where $w_k$ is the *prediction* and $w_1^{k-1}$ is the *history* $w_1 w_2 ... w_1^{k-1}$. Using this notation, the probability of producing a specific string in the language $w_1^k$ can be computed as follows:

$$P(w_1^k) = P(w_1)P(w_2|w_1)...P(w_k|w_1^{k-1}) \tag{2.1}$$

Therefore, a language model is a computational method that is able to estimate the conditional probabilities in (2.1) [15].

### N-gram Language Model

A common example of language model is the *n-gram language model*, which estimates the likelihood of the next word in a sequence as dependent only on the N-1 preceding words [63].

To achieve this estimation, the n-gram model considers the probability of an entire sequence of words, defined as follows.

$$P(X_1 = w_1, X_2 = w_2, ..., X_n = w_n) = P(w_1, w_2, ..., w_n) \tag{2.2}$$

The preceding joint probability can be rewritten using the chain rule:

$$\begin{aligned} P(X_1, X_2, ..., X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1,2})...P(X_n|X_{1,...,n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_{1,...,k-1}) = \prod_{k=1}^{n} P(w_k|w_1, ..., w_{k-1}) \end{aligned} \tag{2.3}$$

Therefore, the joint probability is composed as a production of the conditional probabilities of the entire history. However, computing the sequence of conditional likelihoods is demanding. To overcome this limitation, the n-gram models assume that the likelihood of a word can be approximated with only a fixed window of previous N-1 words of the history. This hypothesis is known as Markov assumption.

Therefore, the joint probability of the entire word sequence having N=2 (2.3) can be rewritten using the probability approximations of each word in the sequence:

$$P(w_1, w_2, ..., w_n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{2.4}$$

To understand the probability estimation (2.4), we consider the following example. Given a bi-gram model (i.e., a n-gram model having N=2), we aim to compute the joint probability of the sequence "The sun is yellow".

The sentence is split in words as follows: $w_1 = "The"$, $w_2 = "sun"$, $w_3 = "is"$, $w_4 = "yellow"$). Therefore, the probability approximation of the sentence is:

$$\begin{aligned} &P(The, sun, is, yellow) \approx \\ &\qquad \approx P(The|<s>)P(sun|The)P(is|sun)P(yellow|is)P(</s>|yellow) \end{aligned} \tag{2.5}$$

Where the symbols <s> and </s> represent respectively beginning and end of the sentence.

While the importance and usefulness of n-gram model is evident, its inherent weakness must also be highlighted. When the model evaluates a sequence that has not been included in the training set, the probability assigned to that sequence will be equal to zero. For this reason, n-gram models cannot handle sequences that are not evaluated during the training. The problem, however, can occur frequently, as natural languages have large vocabularies and therefore training datasets do not always include the sequence under analysis. This phenomenon is called "curse of dimensionality". Bengio et al. [13] overcome this problem proposing the application of neural networks to language models.

Due to the versatility and performance achieved, different language models were developed to solve various tasks, such as machine translation, natural language generation and speech recognition. In next section we will give an overview of the application of neural networks to natural language processing.

## 2.2 Deep Neural Networks for Natural Language Processing

In this section we provide an overview of different neural networks applied to natural language processing tasks. We start the analysis with basic architectures, such as the feed-forward and recurrent neural networks, and then we progress to more complex ones, including the transformers.

### 2.2.1 Feed-Forward Neural Network

Neural network architectures are characterized by different number and type of layers used, interconnections and parameters. Two of the most popular architectures are feed-forward neural networks (FNNs) and recurrent neural networks. These two approaches are differentiated by the way data are flowing through their architectures.

In feed-forward neural networks, the input information is *forwarded* through a network with no loops. Starting from the input layer, the information is evaluated by multiple hidden layers, until reaching the output layer. Recurrent neural networks, instead, are more complex architectures as they are characterized by bi-directional flow of data. In this paragraph we will give an overview of the first approach, starting from a basic FNN architecture, then expanding the analysis to the application of FNNs in language modeling.

#### Feed-Forward Architecture

A feed-forward network is composed by a set of neurons, interconnected together and ordered in layers. Each neuron $x_i$ is defined by a *mapping function* $\Gamma$, assigning to the neuron its ancestor nodes $\Gamma(x_i) \subseteq \mathrm{V}$, and its predecessor nodes $\Gamma^{-1}(x_i) \subseteq \mathrm{V}$. Each neuron is also characterized by a threshold coefficient $\Upsilon_i$ and it is connected with neurons of the next layer. Every connection between neurons $x_i$ and $x_j$ is associated to a weight $\omega_{ij}$, representing its importance in the architecture [117].

A feed-forward neural network is defined as *fully connected* if every neuron is linked to all neurons of the next layer, and *partially connected* otherwise [120].

Given $\xi_i$, that is the potential of neuron $x_i$, and the activation function $f(\xi_i)$, the output of a neuron is determined by:

$$x_i = f(\xi_i), \quad \xi_i = \Upsilon_i + \sum_{j \in \Gamma_i^{-1}} \omega_{ij} x_j \tag{2.6}$$

During the supervised training phase, the model tries to minimize the objective function, which measures the difference between the actual result and the expected one. This is achieved by updating threshold coefficients $\Upsilon_i$ and weights $\omega_{ij}$ of the neuron $x_i$.

Because of its versatility, feed-forward neural networks architecture has been successfully applied in many fields, including also the natural language processing. The following section is therefore focused on the usage of FNNs in language modeling.

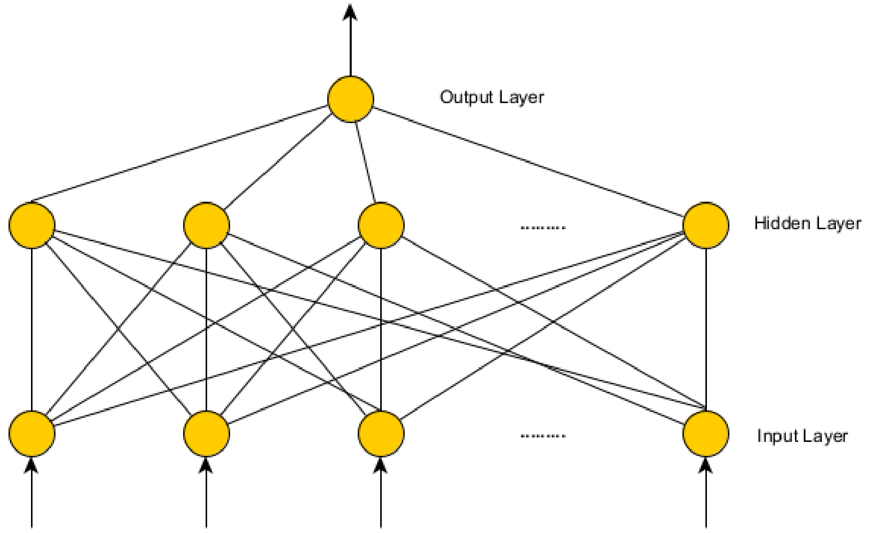

**Figure 2.1:** Example architecture of feed-forward neural network provided in [117]

### Feed-Forward Neural Language Model

Bengio et al. [13] propose the first example of a neural network applied to language models. As mentioned in previous sections, statistical language models try to learn the joint probability function of word sequences in a language. The task, however, is challenging due to the curse of dimensionality. To overcome this issue, the research implemented language modeling through feed-forward neural networks.

The approach creates *distributed word feature vectors* (also known as embedding vectors), each one representing a word in the vocabulary. Embedding vectors can be seen as coordinates of a point in a vector space, representing the word being embedded. The purpose of these vectors is to represent all meaningful features of words in a lower-dimensional space.

In the research, Bengio et al. use the word feature vectors for rewriting the joint probability learned by the language model. Furthermore, to maximize the required log-likelihood during training, and therefore minimize the perplexity, the researchers also propose to incrementally update the parameters associated to probability functions being learned. This process allows to learn concurrently both the distributed representations of words and the parameters associated to the probability function.

The use of feature vectors also facilitates the model generalization of words being analysed. In fact, when the language model learns the representation of semantically (or syntactically) similar words, is also learns how to represent that similarity in the corresponding embedding vectors. Using this approach, the language model is able to understand when a word of an unseen sequence is similar to a word included in a sequence analysed during training phase. This result is fundamental for the future researches in natural language processing field, as it allows to compare the representations of words being processed by a model.

### Language Model Definition

Given a finite vocabulary V, a length T, and a word sequence $WS = w_1...w_T$ for $w_t \in V, \forall t \leq T$, the model aims to learn the function:

$$f(w_t, ..., w_{t-n+1}) = P(w_t|w_1^{t-1}) \tag{2.7}$$

This function can be expressed with a mapping $C$ and a mapping $g$. The mapping $C$ represents the distributed feature vector. It associates any word in the vocabulary $i$ to a vector $C(i) \in \mathbb{R}^m$. The function $g$, instead, maps an input sequence of feature vectors $(C(w_{t-n+1}), ..., C(w_{t-1}))$ to a conditional probability distribution over V for the next word $w_t$. Therefore, the function $f$ can be rewritten in terms of $g$ and $C$:

$$f(w_t, ..., w_{t-n+1}) = g(i, C(w_{t-1}), ..., C(w_{t-n+1})) \tag{2.8}$$

Where the mapping C is a matrix $|V| \times m$, having size of the feature vector $m$ and size of the vocabulary $|V|$. The model learns the parameter C during the training. Concatenating all feature vectors in WS, we can express the following vector:

$$x = (C(w_0), C(w_1), ..., C(w_T)) \tag{2.9}$$

This is given as input to the hidden non-linear layer $y$. Futhermore, given:
- $h$, the number of hidden units
- $m$, the number of word features
- $b$, the output bias of size $|V|$
- $U$, the matrix of hidden-to-output weights of size $|V| \times h$
- $tanh$, a non-linear activation function called hyperbolic tangent
- $d$, the hidden layer biases, composed by $h$ elements
- $H$, the matrix of hidden layer weights with dimension $h \times (n-1)m$

The output of hidden non-linear layer is defined as:

$$y = (b + Utanh(d + Hx)) \tag{2.10}$$

The output $y$ is then given in input to a softmax output layer:

$$P(w_t|w_{t-1}, ..., w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}} \tag{2.11}$$

The softmax function is often used in final layer of neural networks for representing their outputs as probabilities. When softmax function is applied on an input vector, the components of resulting vector will be in the interval (0,1) and their sum will be equal to 1. Therefore, the output is interpreted as a probability distribution, where each component represents the probability associated to a particular class among all the available classes. In language models, each class represents a word of the vocabulary.

### Training

Given the parameter set $\Theta = (C, \omega)$, where $\omega$ are the network's parameters, the training aim to find $\Theta$ that maximizes the following log-likehood:

$$L = \frac{1}{T}\sum_t logf(w_t|w_{t-1}, ..., w_{t-n+1}; \Theta) + R(\Theta) \tag{2.12}$$

Where $R(\Theta)$ represents the regularization term. The training is performed by applying Stochastic Gradient Descent and Backpropagation algorithms [61].

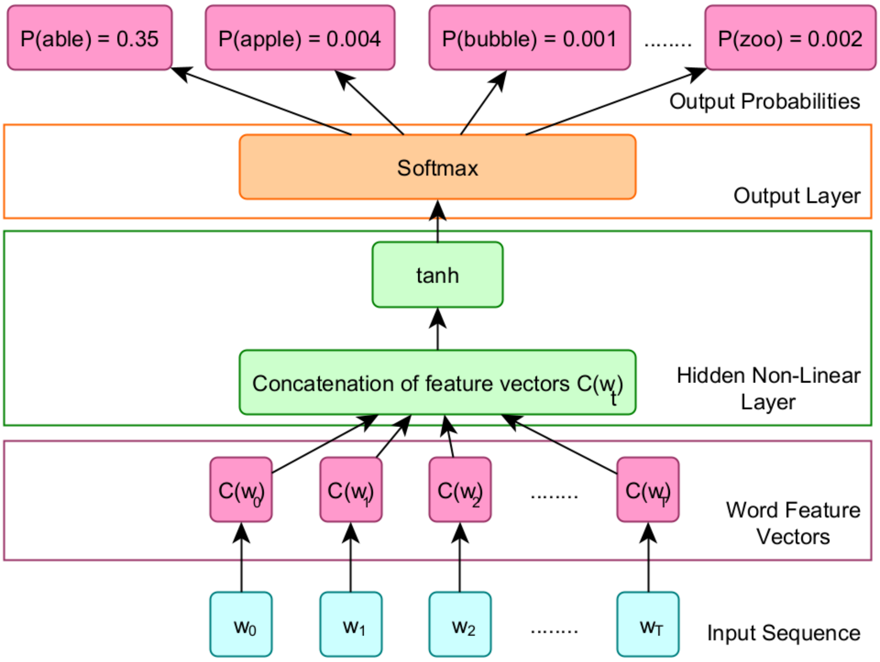

**Figure 2.2:** Architecture of feed-forward neural network proposed in [13]

### 2.2.2 Recurrent Neural Network

In the previous section we have analyzed FNNs, architectures that forward data one-way through all their layers. Due to their acyclic architecture, FNNs are not able to use the outputs of intermediate layers to influence the computations of subsequent layers. Furthermore, these architectures can only work with fixed length contexts and cannot handle sequences of arbitrary size. These constraints are addressed in recurrent neural networks, that allow to use loops and sequences of arbitrary size. Due to their features, RNNs are applied in many fields, including text and speech recognition [1, 109].

RNN architectures can also simulate dynamic systems with discrete time $t$. However, the complexity of the system being modeled influences the optimal number of hidden layers required for accurate representation. For this reason, Pascanu et al. [93] show different ways to implement deepening in recurrent neural networks. The first method consists of increasing complexity of the function transforming input information into the hidden state. The second method, instead, increases the complexity of hidden-to-hidden transition, for example by adding multiple hidden layers. Finally, the third approach increases the complexity of the transition between hidden layer and output.

#### Model Definition

Given a sequence of inputs $S_t = \{..., \mathrm{x}_{t-1}, \mathrm{x}_t, \mathrm{x}_{t+1}, ...\}$, where $\mathrm{x}_t = (x_1, x_2, ..., x_N)$, a matrix of weights $W_{IH}$ representing the connections between input layer and hidden layer, M hidden units $h_t = (h_1, h_2, ..., h_M)$, and P output units $Y_t = (y_1, y_2, ..., y_P)$.
The dynamic system over discrete time $t = (1, ..., T)$ is defined as:

$$h_t = f_H(o_t) \tag{2.13}$$

$$o_t = W_{IH}\mathrm{x}_t + W_{HH}h_{t-1} + b_H \tag{2.14}$$

$$y_t = f_O(W_{HO}h_t + b_O) \tag{2.15}$$

Where $f_H$, $f_O$ are activation functions for hidden and output layers, while vectors $b_H$,$b_O$ are respectively biases of hidden units and output layer. Finally, the matrices $W_{HH}$, $W_{HO}$ represent connection weights between hidden layers and between hidden and output layers.

This representation is iterated over time $t$, and the hidden layer $h_t$ is representing the *memory of the system*. The activation functions introduce non linearity, permitting to model complex systems that can be used successfully also in natural language processing.

The architecture of recurrent neural networks cannot properly handle long-term sequential information as input [14]. This limitation, called *vanishing gradient descent*, is partially solved in long-short term memory RNNs.

### Recurrent Neural Language Model

In the following we analyze the application of RNNs in language modeling. Mikolov et at. implement a language model using a recurrent neural network [89]. The architecture is composed by identical cells, one for each element in the input sequence of words. At each time $t$, the input $x_t$ of the RNN is concatenating both the embedding vector of current word of the sequence $C(w(t))$, and the output of previous iteration $s(t-1)$. Therefore, the language model produces two outputs: $s(t)$ and $y(t)$. The first vector is representing the memory of the model and it is given as input for next time step. The latter is instead the output vector, forwarded to next layers of the network. The language model is defined as:

$$\mathrm{x}(t) = [C(w(t))^T s(t-1)^T]^T \tag{2.16}$$

$$s_j(t) = f(\sum_i x_i(t)u_{ji}) \tag{2.17}$$

$$y_k(t) = g(\sum_j s_j(t)v_{kj}) \tag{2.18}$$

Where $f$ and $g$ are respectively the sigmoid $f(z) = \frac{1}{1+e^{-z}}$ and softmax functions.

The recurrent neural network is trained by minimizing the following cross-entropy loss function at each time step $t$:

$$L(t) = -log(y_{w_(t+1)}(t)) \tag{2.19}$$

Where $y_{w(t+1)}$ represents the probability of the word $w_{t+1}$ at the time step $t$. Furthermore, the training phase updates the weights of the RNN using backpropagation algorithm.

### Bidirectional Recurrent Neural Network

As showed in preceding sections, during training the recurrent neural networks analyze only information regarding previous iterations. To overcome this limitation, Schuster et al. [111] proposed a bidirectional recurrent neural networks (BRNNs), which analyzes both future and past information in each time step.

The architecture is composed by two different recurrent neural networks. The first RNN represents forward states and it evaluates inputs in the positive time direction. The second RNN is instead representing backward states and it analyzes the information in the negative time direction (i.e., use reverse direction). Furthermore, the output of an RNN is never given as input to the other one. The output of the bidirectional neural network is obtained by concatenating the outputs of the two states.

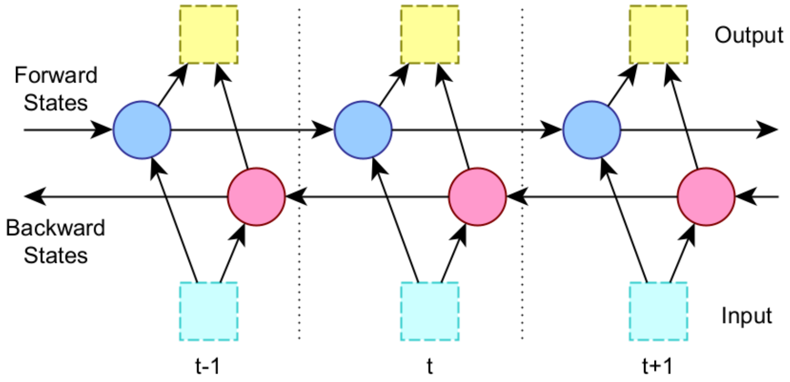

**Figure 2.3:** Architecture of bidirectional neural network proposed in [111]

### Long-Short Term Memory Recurrent Neural Networks

The long-short term memory recurrent neural networks (LSTM RNNs) were introduced by Hochreiter et al. in [60]. The method reduces the limitations caused by vanishing gradient descent. The hidden units are considered as memory cells that remember an information. Their input and output flows are controlled with specific units, known as input and output gates. A third unit, called forget gate, were proposed by Gers et al. [47]. The forget gate allows to select which information should be deleted in a preceding state.

## 2.2.3 Word2vec

Word2vec are popular models used for learning high-quality word embedding, and they were proposed for the first time by Mikolov et al. in [88]. These models are used in NLP field, as their approach is able to manage word embeddings for large datasets and vocabularies. Furthermore, word2vec is often used in combination with other architectures to build complex language models.

In order to minimize complexity of word2vec models, researchers decided to not include any non-linear hidden layers in their architectures. As a consequence, word2vec can only learn distributed word representations of fixed length sequences.

The researchers proposed two different architectures for word2vec: *continuous bag-of-words model* (CBoW) and *continuous skip-gram model.*

The first architecture allows to predict a central word given the surrounding context of words. The name of this architecture derived from the fact that word embeddings does not depend on the position of words in the history. The CBoW model creates embeddings of word sequences in input and compute an average of the obtained feature vectors. Then, the embedding vectors are forwarded to a linear layer and to a softmax layer. This final step is used to represent the outputs as probabilities.

The second architecture uses an opposite approach with respect to CBoW. The task of Continuous Skip-gram Model is to predict the surrounding context of a given word. The model creates the feature vector of the current word, and then give it as input to a linear layer. The output is used to predict the context to the left and the right of the current word.

### Word Embedding Evaluation

Mikolov et al. [88] propose an approach for evaluating the quality of word embeddings. The evaluation is performed using the concept of syntactic and semantic similarity relationship between word pairs. Given a pair of known similar words and an unmatched word, the goal is to find the similar word to the unpaired one. For example, given the known word pair "loud", "loudest", and the unmatched word "quiet", the task is to find the word replying to the following question: « What is the word that is similar to 'quiet' in the same sense as 'loudest' is similar to 'loud'? ». To answer the question, researchers compute algebraic operations with vector embeddings of the considered words:

$$X_{vector} = loudest_{vector} - loud_{vector} + quiet_{vector} \tag{2.20}$$

When X is computed, the closest word to X is searched in the vector space by computing the cosine distance between vector X and the embedding vector of the analyzed word. The result is then selected to answer the above question. If the model has learned a good quality vector representation, the word selected will be the correct answer (that is, in the previous example, the word 'quietest'). Therefore, word2vec word embedding represents similar words as close vectors in the vector space. This result is very interesting as it allows to compare different embeddings in their vector space.

To test word2vec model, a dataset of semantic and syntactic questions (i.e. word pairs) is created. The evaluation has been performed considering the ratio between correct and wrong answers replied using the word embedding provided by word2vec. Researchers compare the results of different architectures: recurrent neural language model, feedforward language model, continuous bag-of-words model, and continuous skip-gram model. The experiment shows that word2vec outperforms the other models. Skip-gram correctly replies to 55% of semantic similarity questions, CBoW returns 24% correct answers, feedforward and RNN language models give, respectively, only 23% and 9% correct answers. For syntactic similarity, instead, CBoW replies correctly to 64% questions, skip-gram gives 59% of correct answers, while feed forward and RNN language models answering correctly to 23% and 9% questions respectively.

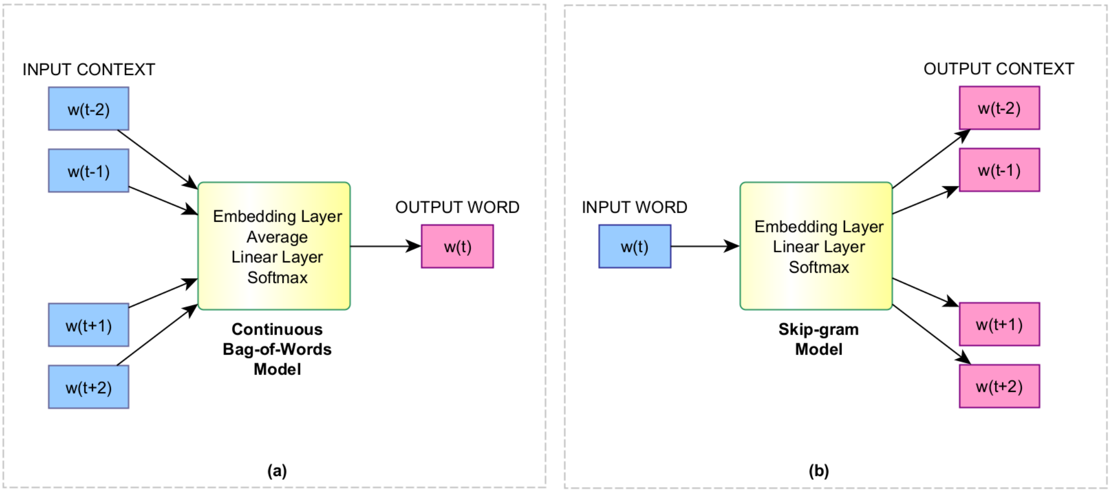

**Figure 2.4:** (a) Continuous Bag-of-Words Model and (b) Skip-gram Model architectures of Word2vec [88]

### 2.2.4 Transformer

Trasformers are deep learning models proposed by Waswani et al. in [121], and firstly used for solving NLP tasks. These models, however, can achieve great performance in different fields, such as computer vision [53] and speech recognition [37]. In addition, they are used as basis for complex architectures, reaching state-of-the-art performance in different tasks.

One of the most important elements characterizing the transformer architecture is the *self-attention mechanism*. This type of attention allows the model to focus on most relevant words in a processed sequence, and capture relationships between words in the sentence. Using this mechanism, the model is able to reach state-of-the-art performance in different NLP tasks.

#### Architecture

The transformer model described in [121] is built as an encoder-decoder architecture. The encoder first represents the input sequence with a continuous representation. This is then given in input to the decoder component for producing its output.

The *encoder component* consists in N identical layers, each one divided in two sub-layers: multi-head self-attention and fully connected feed-forward network. After each sub-layer are added residual connection and layer normalization. The first is needed to overcome vanishing gradient descent problem (described in [2.2.2]), while the latter helps in stabilizing the training phase.

The *decoder component* also comprises N identical layers. Each layer is composed by three sub-layers: masked multi-head self-attention, multi-head self-attention, and fully connected feed-forward network. The masking is used to guarantee that predictions are only dependent on previous outputs and not the next ones. For the same reason, an offset is applied to output embeddings. Finally, to each decoder's sub-layer is applied residual connection and layer normalization.

### Multi-Head Attention and Output Probabilities

The Transformer architecture uses a *scaled dot-product attention* function. This mechanism maps an input composed by a query and a key-value pair to an output. In detail, given the matrices:
- Q, a set of queries regarding which information the model is trying to understand
- V, consisting in values associated to each word in the sentence
- K, including the keys for values in V
The output scaled dot-product attention is computed as:

$$Attention_{Q,K,V} = softmax(\frac{QK^T}{d_k^{1/2}})V \tag{2.21}$$

Where $d_k$ is the dimension of matrices Q and K, while $d_v$ is the size of matrix V.

To increase their ability to learn relationships between words, transformers use multiple (i.e., "multi-head") attention functions. Therefore, queries Q, values V, and keys K are linearly projected h times, where h is the number of heads used. On each of these projections is then applied the scaled dot-product attention function (2.21).

In this phase, each attention head computes different weights, representing different levels of importance assigned to their inputs. The output of multi-head attention is computed as a linear transformation of the concatenation of single-head outputs.

The output of encoder component is then given in input to a linear layer, and a final softmax layer is used to transform decoder outputs into probabilities of next-token predictions.

### Positional Embeddings

Transformers model learn word embeddings for representing input and output tokens into vectors. Transformers use *positional encoding* representing token's position in a sequence and having the same size as the word embedding *d*. The encodings are summed to input and output embeddings. The resulting vectors are injected in the encoder-decoder architecture.

### Training

Researchers trained the proposed model on two datasets for language translation. The first is composed by 4.5 million pairs of English-German matching sentences, while the second one includes 36 million of English-French pair sentences.

The datasets are encoded using an algorithm for word sequence tokenization known as *byte-pair encoding.* Using this encoding, researchers define two vocabularies, one for each of the two datasets. Furthermore, sentence pairs of similar lengths are grouped into batches and used to train the model. During training, dropout and label smoothing regularization are applied to both encoder and decoder components. The dropout is applied to outputs of the sub-layers, and outputs of the sums between embeddings and positional encodings.

Researchers trained two different configurations of the transformer: a base model having $d$=512, and a big model with $d$=1024. Transformer architecture has been tested on the task of English consistency parsing. The experiments show great performance of the transformer with respect to sequence to sequence architecture [11].

### Transformer-Based Architectures

The adoption of the self-attention mechanism in transformers has improved their representation and generalization capabilities, making the models usable in a wide range of tasks. These important improvements have resulted in transformers being used as the basis of well-known large language models, such as GPT [2, 16, 57] and BERT [33]. These architectures are generally pre-trained as general purpose models, and then fine-tuned on specific tasks. For example, BERT has been fine-tuned for various tasks in different fields, such as code representation (e.g., CodeBERT [41], graphCodeBERT [51], CuBERT [64]), translation (e.g., RoBERTa [83]), biomedical language representation (e.g., BioBERT [75], Clinic BERT), and mono-language understanding tasks (e.g., alBERTo [99], Arabert [8], Bertje [32])

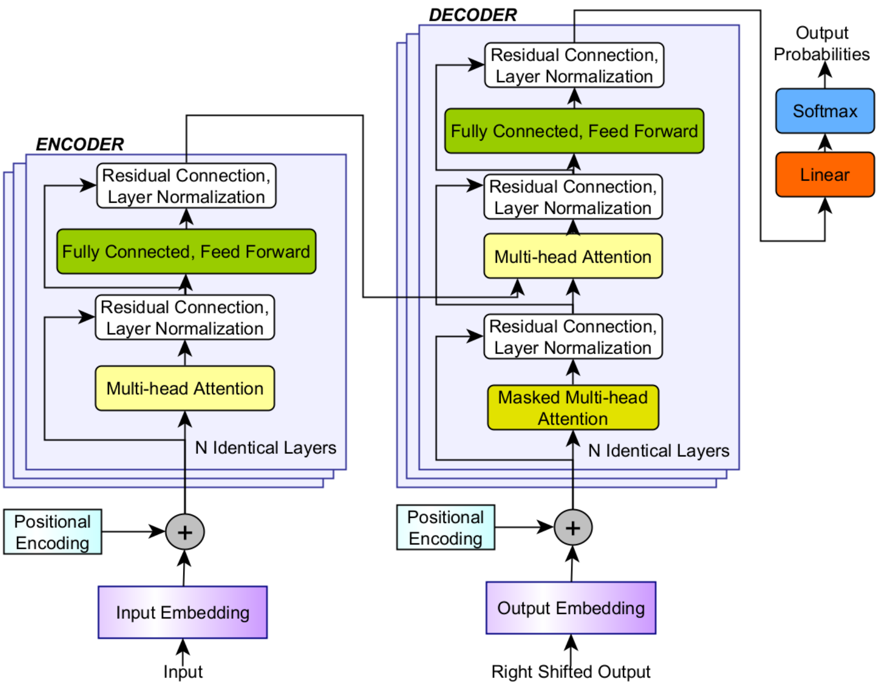
ENCODER
Residual Connection, Layer Normalization
Fully Connected, Feed Forward
Residual Connection, Layer Normalization
Multi-head Attention
N Identical Layers
Positional Encoding
+
Input Embedding
Input
DECODER
Residual Connection, Layer Normalization
Fully Connected, Feed Forward
Residual Connection, Layer Normalization
Multi-head Attention
Residual Connection, Layer Normalization
Masked Multi-head Attention
N Identical Layers
Positional Encoding
+
Output Embedding
Right Shifted Output
Output Probabilities
Softmax
Linear

**Figure 2.5:** Transformer architecture proposed in [121]

# Chapter 3

# Background on Binary Code Analysis

This chapter gives an overview of various methodologies used for binary code analysis based on neural networks. The chapter also provides a description of the programming paradigms and compilation process, to understand how the binaries are generated.

For the binary code analysis, we study a specific subset of DNN architectures that learn the representation of binary functions for solving various tasks. This study allows to understand meaningful aspects of the code and, therefore, it gives a deeper insights into the overall software functionalities. In order to perform the analysis, researchers adopted the same deep learning techniques used for language models. In fact, as discussed earlier, the "naturalness hypothesis" [3] posits a correlation between code and natural languages. This connection justifies the successful application of DNNs, originally designed for NLP, to the code analysis. As a matter of fact, language models have shown great success in learning the relationships between words in a sentence. Similarly, the application of DNNs techniques to code, even in its binary form, allows to learn relationships between parts of the software, compare different code snippets, and also understand information on the code functioning. Recent researches on binary code analysis demonstrate that the application of this approach is able to achieve state-of-the-art performance in various tasks, such as similarity between binary functions, and recovery of original function parameters.

The chapter is split in two main parts. The section 3.1 provides a background on programming. It introduces paradigms and formal definitions of programming languages through grammars. This part also details the process of code compilation and disassembly of binary code. The section 3.2 analyzes various models for binary code analysis, with a special focus on DNNs solving binary function representation tasks.

## 3.1 Background on Programming

The 1940s saw the introduction of the first electronic computers. These innovative tools were characterized by their ability to compute simple arithmetic operations and the possibility of be programmed using low-level machine languages [45]. These programming languages, known as *first generation languages*, were composed by simple instructions that could be executed directly on the processor. While these capabilities seem basic by today's standards, they represented the beginning of human-computer interaction and set the stage for the future of computing. A further advance was represented by assembly languages, considered *second generation languages.* These languages introduced a level of abstraction with respect to machine languages, by using symbols to express operations and operands. This symbolic representation is not directly understood by machines, but need to be translated through a specific program, known as an assembler. Each machine's architecture, however, has a specific Assembly language. Therefore, programs written in these languages cannot be reused in different architectures. In the 1950s, *third generation languages* were introduced. These high-level languages are independent of the architectures used, and allow more human-readable instructions to be written. This approach made programs portable across different machines. Due to their high-level nature, these languages require to be translated into machine language to be executed using either interpreters or compilers.

The evolution of programming languages brought with it a variety of programming paradigms, each emphasizing distinct aspects of program design [107]. These represent fundamental approaches for structuring computer programs. However, we can define two principal categories of paradigms: imperative and declarative [112]. The *imperative paradigm* explicitly defines the set of instructions to be executed, controlling the execution flow of the program. These instructions allow to modify values of variables through side effects, impacting the outcomes of calculations [12]. Examples of imperative programming languages are Fortran, Java, Pascal, C, C#, C++, and Python. Among imperative paradigms, there are procedural programming, focusing on procedure calls, and object-oriented programming, operating on objects. These objects contain data and code, and interact with each other to obtain the required outcome [45]. In contrast, *declarative paradigm* focuses on describing the logic of the computation and the attended outcome, without explicitly defining the execution flow of programs. Examples of declarative programming are: Prolog, Datalog, and XML, and HTML. This approach includes various sub-categories, such as logic programming, based on formal logic, and functional programming, focused on the definition of functions. The invocation of these functions is characterized by the absence of side effects. The output of each function call is determined only by the input arguments [18]. There is no absolute best programming paradigm. Each approach has its own characteristics and solves problems concerning a specific context. Programming languages can be aligned with one or more paradigms. An example of these languages is Python, supporting both object-oriented and functional programming.

### 3.1.1 Description of Programming Languages

Within the domain of computer science, programming languages represent formalisms for expressing instructions that computers execute. Analogous to natural languages, programming languages are described using grammars. We can consider a grammar as a group of rules defining how to combine characters from an alphabet to create valid strings in a specific programming language [125]. In detail, a *formal grammar* is quadruple of four components $G = (N, T, P, S)$. T is a *terminal alphabet*, finite and non empty set of characters used to compose words of the language. N is a *non terminal alphabet*, a finite and non empty set of non terminal symbols. P is a set of *production rules*, defining how to substitute non terminal symbols with terminals and/or non terminals. Last, the *start symbol* $S \in N$ is the initial point for generating valid strings using production rules in P.

Using a similar notation, we define the formal language generated by a given grammar G as follows: $L(G) = \{w \in T^* | S \implies {}^*w\}$, where $T^*$ indicates all finite strings of characters in the terminal alphabet T, while ${}^*\!\!\implies$ represents a finite sequence of substitutions using production rules in P [45, 80].

Chomsky hierarchy [21] classifies language grammars into four categories based on their complexity, namely Type 3, Type 2, Type 1, and Type 0. Type 3 represents *regular grammars*, the simplest among the hierarchy. They describe production rules in the form: $S \implies xS$ and $S \implies x$, or $S \implies Sx$ and $S \implies x$, where S is the start symbol and x $\in$ T. These grammars are recognized by finite state automata. Type 2 instead are the *context-free grammars.* These are a super set of Type 3, recognized by pushdown automata. Context-free grammars uses production rules of the type $S \implies \gamma$, where $\gamma$ represents strings of terminal and/or non terminal symbols. Type 1 are the *context-sensitive grammars.* These include Type 2 grammars, and have production rules in the form: $\alpha S\beta \implies \alpha\gamma\beta$, where $\alpha, \beta, \gamma$ are sequences composed by terminals and/or non terminals. These grammars are recognized by linear bounded automata. Finally, Type 0 are *unrestricted grammars*, the most powerful of the hierarchy. These include Type 1 grammars and use production rules in the form $\alpha \implies \beta$, where both $\alpha, \beta$ are sequences of non terminal and/or non terminal symbols, and $\alpha \neq \emptyset$. Type 0 grammars are recognized by Turing machines. A detailed description of the mentioned automata is provided the Appendix A.

In the context of computer science, programming languages can be described mostly with context sensitive grammars. However, context-free grammars are widely used tools to express the structure, or *syntax*, of programming languages. These grammars can represent potentially all constructs of modern programming languages. Also, the syntax of context-free languages can be efficiently analyzed using parsing algorithms that are

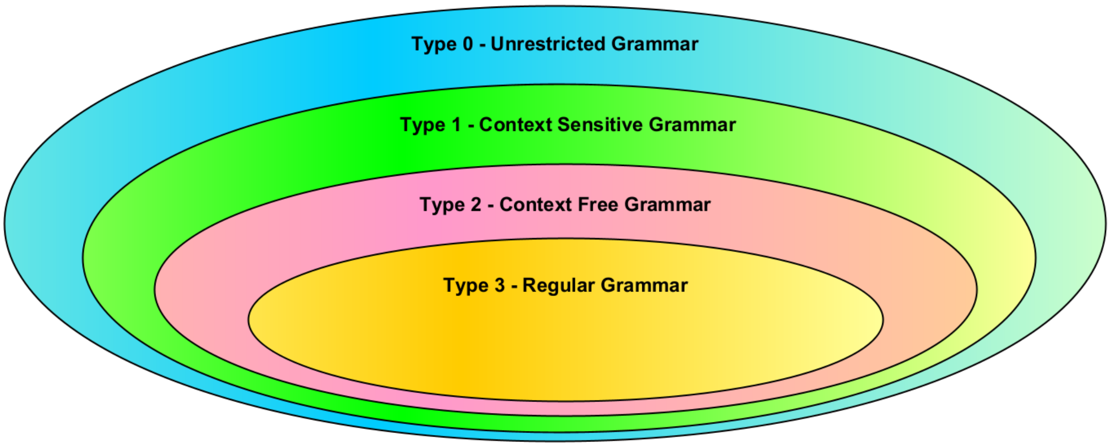
Type 0 - Unrestricted Grammar
Type 1 - Context Sensitive Grammar
Type 2 - Context Free Grammar
Type 3 - Regular Grammar

**Figure 3.1:** Chomsky hierarchy of language grammars

implemented in present compilers. As a consequence, context-free grammars are widely used tools for representing programming languages [116].

### 3.1.2 Program Compilation

The process of translating source code into a form executable by the target architecture depends on the programming language used [12, 50]. For Assembly programming language, this process is performed by a program called assembler. It operates a one-to-one translation of programs and data into binary format. The assembler also manage the conversion of internal and external addresses referenced in the code. For high-level programming languages, the process is called compilation and it is performed by a program known as *compiler*. This process is responsible for translating source code into a machine-specific executable format. A different compiler will be employed depending on the programming language and architecture used to run the code.

The structure of a compiler is divided in two sections: front-end and back-end. The *front-end* examines syntax and semantics of a program with respect to the grammar of the programming language. It is divided in several modules. The first module takes the program's file in high-level programming language and, along with other necessary files, transforms its content into a sequence of characters. The next module, focused on lexical analysis, translates the extracted characters into tokens and assigns meaning to them. At this point, the parser module performs syntax analysis of the tokens. It constructs the abstract syntax tree (AST) following the grammatical rules of the programming language. The next module is focused on context handling. It annotates AST nodes with additional information regarding the program's context. The last step of the front-end is a generator of the code's intermediate representation, that generally includes control flow instructions and expressions.

The *back-end* of the compiler takes as input the intermediate code and outputs the target machine code to be executed. In detail, the first module of the back-end takes as input the intermediate code and process it for optimization purposes. During this phase, the compiler also insert the code of certain functions directly where they are called. This process, known as in-lining, reduces the time overhead of function calls. The next module is the code generator, converting the AST into ordered symbolic instructions and allocating the required registers. The next module is focused on target-code optimization. It performs a further processing of the symbolic instructions. The machine code generator then translates symbolic instructions into a bit pattern. It defines data and addresses of the code. It also generates in output relocation and constant tables. The final step outputs the object file, including constant and relocation tables, headers, and any essential data for the system to run the program. In addition to generating object files, many compilers, including GCC and Clang, allow to output code in Assembly language during compilation by using a specific flag.

While compilers translate source code into machine code before execution, *interpreters* executes the code without converting it into machine language. An advantage of this approach is that interpreters are architecture-independent and can be used on different machines. In practice, most interpreters, however, do not execute the code directly. The code is in fact first translated into an intermediate code and then executed by the interpreter. An example of this approach is javac, the compiler of the Java programming language. The program compiles the source code in Java language, contained in a .java file format, for an abstract machine. This process translates the code into bytecode (i.e., the .class format). The interpreter, known as Java Virtual Machine, read the bytecode

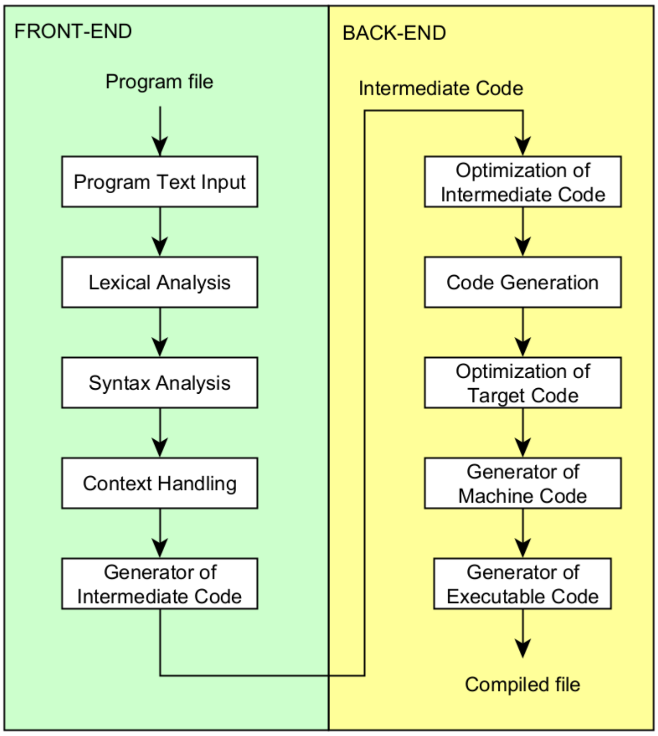
FRONT-END
Program file
Program Text Input
Lexical Analysis
Syntax Analysis
Context Handling
Generator of Intermediate Code
BACK-END
Intermediate Code
Optimization of Intermediate Code
Code Generation
Optimization of Target Code
Generator of Machine Code
Generator of Executable Code
Compiled file

**Figure 3.2:** Modular structure of compilers

and execute it. This process is very useful for running applets in browsers, because it allows code to be executed independently of the the computer's architecture running it.

Programs are usually composed by multiple files for better organization. As a consequence, a single object file generally does not include all the functionalities required for executing the code. Each file in fact contains various modules, and may reference external functionalities from other files. Depending on the programming languages, programs can also import a variety of external libraries. For this reason, after the compilation or the assembling process, the *linker* resolves external references and includes them in a single and self-contained binary file. This process, called static linking, can be slow for complex programs with many files. To address this issue, operating systems often use a technique called dynamic linking. This approach resolves external references at runtime, leaving unresolved symbols in the compiled code. Binaries processed by the linker and the required libraries are loaded in memory using a part of the operating system, called *loader*. This also initializes a stack and a set of registers for executing the program.

In this section we have analyzed how to translate programs into a binary form to be executed on a computer, illustrating the difference between compiler and interpreter. We have also explained the importance of linker and loader. In the following section, we give an overview of decompilation and disassebling processes. These tools try to retrieve the code having generated the binaries in input.

### 3.1.3 Decompiler and Disassembler

Decompiler and disassembler are fundamental tools used for reverse engineering of computer programs. Decompilers have a structure similar to compilers, divided into three modules [23]. The front-end module loads the machine code in memory and parses it starting from the program's entry point, provided by the loader. The module's output is a low-level representation of the binary code and the related control flow graph. The universal decompiling machine performs data flow analysis and control flow analysis of the low-level code stored in memory. This module outputs a high-level intermediate representation of the program. The back-end module then reconstructs the code in high-level programming language using the control flow graph.

Disassemblers, on the other hand, translate binary code into assembly language. There exist two main approaches for disassembling code: *static* and *dynamic*. Static disassembly recovers assembly code without executing the machine code, allowing to recover assembly code of the entire program [72, 73].

Static disassembly uses two main techniques, namely linear sweep and recursive traversal. The linear sweep translates instructions one at a time starting from the beginning of the program's code section. This technique is employed by many disassemblers, including objdump. The linear sweep is, however, not able to distinguish between instructions and data, causing disassembly errors. This limitation is addressed by the recursive traversal technique, reconstructing the assembly code by following the program's control flow from its entry point. When target addresses of jumps or calls are determined at runtime, however, recursive traversal assemblers cannot evaluate those parts of the code. This issue is addressed by speculative disassembly, a technique that applies linear sweep to also recover these unknown sections.

Dynamic disassemblers take the opposite approach, analyzing the program's execution while reconstructing the assembly code. The machine code is typically run within a debugger or an emulator. Dynamic disassemblers are generally accurate for analyzing obfuscated code, but they can only recover the program's parts that are actually executed. As a consequence, static disassembly is generally the more popular choice compared to dynamic one [72].

This first section provided an overview of programming languages, explaining various steps required to execute the code. It also described the decompilation and disassembly, two reverse approaches for reconstructing source and assembly code from binaries. The section provided insight regarding the generation of binary code, which is studied through tools of binary analysis.

The following section presents various approaches for binary code analysis using deep neural networks.

## 3.2 Background on Binary Function Representations

The section presents the most significant DNN architectures for binary code analysis. For a comprehensive understanding of program functionalities, we focus our study on models handling binary functions rather than generic code snippets.

We selected five common tasks to be solved on binary functions and, for each of them, we have provided an overview of the state-of-the-art models. Given their importance, these tasks are also addressed in the benchmark we introduced as first contribution of the thesis.

### 3.2.1 Binary Similarity and Function Search

Binary similarity and function search are two fundamental tasks to be solved on binary functions. Both tasks request to analyze and compare functions represented in their compiled form. Since researches regarding these tasks often overlap, we grouped their related work into a single section.

The problem of binary similarity aims to determine whether two function are similar, according to some definition of similarity, and, usually, to return a similarity score. The task of function search, in contrast, takes as inputs a certain query function and a large database of binary functions. The task requests to identify the $K$ most similar functions to the given query function.

The problem can be solved by using a binary similarity system that computes the similarity between the query and each of the functions that are present in the database. As outcome, the computation returns $K$ most similar functions to the given query. The task of function search is usually presented as an evaluation task for binary similarity solutions, and therefore the literature related to these two problems is essentially the same.

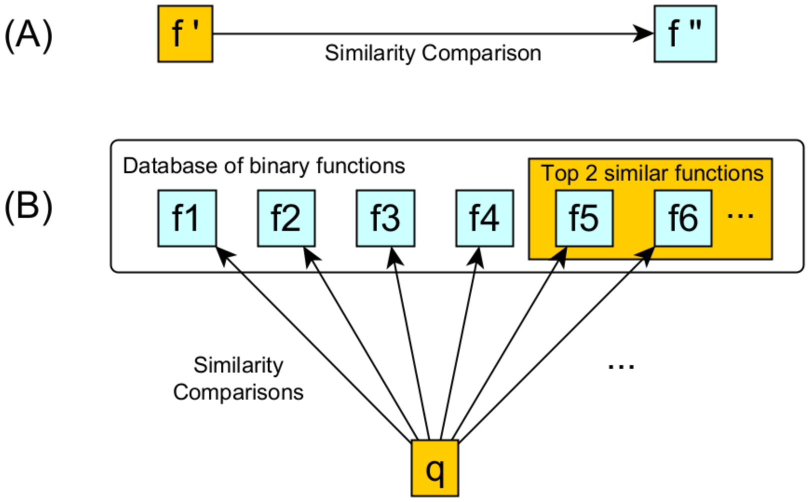

**Figure 3.3:** Difference between binary similarity and function search tasks. (A) Example of binary similarity task. (B) Example of function search task. The problem asks to find K=2 top similar functions to the given query q.

### Binary Similarity

Several studies have explored the problem of binary similarity. For example, [38] proposes to compare executable using an isomorphism between sets of instructions. Khoo et al. [68] introduce Rendezvous, a search engine that detect similar binary code using statistical models. In SIGMA [6], instead, researchers show a method for detecting reused binary functions by comparing their representation in form of Semantic Integrated Graphs.

David et al. [29], address the problem of similarity between functions in stripped binaries. This method split the code in strands and verifies the semantic similarity between code fragments. Another approach to evaluate the similarity between stripped binaries is proposed in [30]. The method decomposed binary functions in comparable units, that are fed in a compiler optimizer to obtain a normalized form of the code. Strands are then compared with statistical techniques.

The model proposed by Xu et al., known as Gemini [129], uses neural networks for solving the task of binary similarity. Gemini transforms each binary function into an attributed control flow graph (ACFG), that is a CFG with manually annotated features. The approach uses a graph embedding network [104] to map the ACFG into a vector. More precisely, each vertex of the graph is assigned to a separate embedding vector. Then, each vertex embedding is updated iteratively considering the vertex features and the graph topology. These vertex embeddings are summed to obtain the representation of the entire function.

The model is trained using Siamese architecture, composed by two identical struct2vec models. Each model takes as input an ACFG and creates the output the corresponding embedding. The architecture then computes the cosine similarity metric between the two output embeddings. The result is compared with ground truth labels, indicating the similarity (i.e., label = +1) or dissimilarity (i.e., label = -1) between function pairs. The goal of the training is to minimize the difference between the predicted similarity score and the actual value. The model has been trained on a dataset composed by 129'365 ACFGs, derived from the compilation of two versions of OpenSSL package using the compiler GCC v5.4, with different architectures and 4 optimization levels. Gemini has been also evaluated on other datasets, to evaluate its performance on specific tasks, such as vulnerability search.

Wang et al. [124] introduced a model for solving the task of binary similarity, called jTrans. The proposed architecture is based on BERT, a transformer model representing the state-of-the-art for a variety of NLP tasks (Details regarding BERT architecture are provided in the section 5.2).

In addition, researchers build and distribute a binary dataset to train and evaluate their model on the task of binary similarity. The dataset is composed by C and C++ project from ArchLinux and Arch User repositories, compiled with GCC and G++ with different optimization levels. The dataset is composed by roughly 26'000'000 binary functions.

The paper proposes a jump-aware representation of binary functions, that includes control-flow information in the code's embedding. The approach disassembles binary functions and tokenizes the resulting assembly codes for normalization purposes. Then, jTrans employs a novel approach to encode positional embeddings and token embeddings, known as parameter sharing. The approach allows to represent the connection between source and target of a direct jump, increasing the degree of similarity between their embeddings. The parameter sharing consists in using the token embedding of the jump source as the positional embedding for the corresponding jump target token. The input embedding of jTrans is therefore obtained as the sum of two components: the positional embedding and the token embedding. This approach allows to use the attention mechanism of jTrans for assigning the same level of attention to both the source token and destination token. This process guarantees that source and destination of a jump are both considered during the evaluation phase.

The research proposes an unsupervised pre-training phase, and then a fine-tuning on the binary similarity task using supervised learning. The similarity between binary functions is evaluated with a similarity score, calculated as the cosine similarity between the corresponding vector embeddings. During fine-tuning, the research tries to achieve two goals: minimizing similarity scores between dissimilar functions, and maximizing similarity scores assigned to similar functions.

The model has also been evaluated on various tasks, including the vulnerability search. In this task, 8 CVEs are selected from a vulnerability dataset. The task requests to search these 8 vulnerabilities in the affected projects, compiled with different combinations of compilers and optimization levels. The metric used to evaluate jTrans on vulnerability search is the recall@10 (i.e., recall@K with K=10). The proposed model shows significant performance, achieving recall of 95% for the task of binary search, and recall@10 of 100% on the vulnerability search.

Qasem et al. introduced BinFinder [100], an approach based on Siamese neural networks focused on the task of searching binary functions. The approach has been trained and evaluated on multiple versions of 7 open-source packages, compiled with GCC and Clang. The approach also uses various obfuscation options, optimization levels and instruction sets. The method extracts two type of features, namely the numerical values and list of literals. These features represents fundamental aspects of binary functions, such as number of callers, number of callees, and names of called functions. The numerical features are converted into lists of sequential values. Then, a set of weights is computed. The resulting features are embedded into one-hot encoding.

The proposed Siamese architecture is composed by a two identical multi layer perceptrons operating in parallel on the selected input features of the same functions. The models are trained to learn the representation of binary functions and generate their vector embeddings. Then, BinFinder computes cosine distance between the two vectors.

### Binary Similarity and Function Search

A number of studies investigate the task of function search as a sub-problem of binary similarity. Research In the work [31], David and Yahav proposed a method for searching functions in stripped binaries using semantic similarity. The approach computes the semantic similarity between tracelets, which are essentially partial segments of a code's execution path. To perform the analysis, the method splits control flow graphs (CFG) of binary functions into these smaller execution snippets called tracelets. Furthermore, the researchers defined a set of rewriting rules for tracelets, allowing to determine the minimum number of modifications needed for transforming a tracelet into another. This approach permits to compute the similarity, and therefore solving function search problem, regardless of code modifications performed by the compiler optimization. The approach is implemented through a tool, which allows to find similar functions in a database composed by a million of binary functions.

Another interesting approach is the one of proposed by Lakhotia et al. [74]. They define the concept of juice, representing an abstraction for the semantics of a binary code. The juice is constructed by first disassemble the binary code, decompose it in blocks and extracting the control flow graph, which shows how these blocks are connected and how the program executes. Then, researchers compute semantics and juice for each block. The semantic of code is defined through symbolic interpretation. The generalization of code's semantics provided by juice, instead, can be retrieved by substituting logical variables to register names, and abstracting literal constants. Researchers implemented the method with a tool, BinJuice. They evaluated the tool on binary similarity and function search tasks, using different datasets. The experiment of function search used different variants of a virus. For binary similarity, instead, the approach has been tested on different versions of a software from GitHub.

Liu et al. [81] proposed $\alpha$Diff, a solution for detecting binary similarity and solving function search task. The work extracts different features from binaries, namely the *intra-function* (raw bytes), the *inter-function* (function calls) and the *inter-module* (library imports). The dataset used for $\alpha$Diff is publicly available and consists of a collection of cross-version binaries from the x86 Linux platform.

The research evaluates the problem of computing similarity score between stripped binaries, obtained from the compilation of different versions of the same source code. The task is composed by two steps. The first is *function matching*, where for each function in a binary it is required to find its correspondent in the other binary. The second step consists in computing the *semantic similarity score* between each pair of matching functions. The score is in the range [0,1], depending on the level of similarity of the functions.

Researchers also evaluates their approach on variants of the main binary similarity problem. The first variant is cross-optimization, where the similarity is computed between binaries obtained from same source code and compiler, but different optimization level. The second variant is cross-compiler, where the two binaries are obtained from different compilers. Finally, the third variant is cross-compiler, consisting in evaluating the similarity between binaries compiled for different architectures.

To evaluate the level of similarity between binary codes, researchers use the concept of function search. Given two binaries $B_1$ and $B_2$, having respectively M and N number of functions, and $V_{B_1,B_2}$ pairs of matching functions where $V_{B_1,B_2} \leq M, N$. For any function $f_{B_1}$ in $B_1$, the solution aims to find top K similar functions to $f_{B_1}$ in $B_2$ (if they exist). For each function $f_{B_1}$ in $B_1$, the solution also computes Recall@K, measuring the number of matching functions appearing within the top K, over $V_{B_1,B_2}$.

The work considers all aspect characterizing a function, by analyzing intra-function, inter-function and inter-module features. In detail, researchers propose to extract *intra-function features*, representing the intrinsic characteristics of a function. The extraction is performed through neural networks. For each binary function, the proposed architecture analyzes raw bytes composing the binary function itself, and generates its vector embedding representation.

The architecture used for the extraction is called *convolutional neural network* (CNN), a specific type of feed-forward neural network that is able to derive features using convolution operations [79]. The architecture of the CNN is composed by 8 convolution layers, followed by 8 batch normalization layers, 4 max pooling layers and 2 fully connected layers. The batch normalization is used to reduce overfitting issues, by normalizing the input coming from previous hidden layer, and passing it to the next layer. Max pooling layers, instead, can be used for reducing the spacial size of the features taken as input [48]. Finally, fully connected layers are generally used to flatten the spatial structure of the input vector.

The model uses activation function Rectified Linear Unit (ReLU) as activation function for the entire architecture. For the training phase, the model learns function embeddings using a Siamese architecture, which is composed by two identical embedding CNNs. The architecture takes as input two functions, one for each CNN, and a binary label representing whether or not the two functions are similar. Then, each CNN generates an embedding for the binary function in input.

Given the network parameters $\theta$, two functions $F_a$ and $F_b$ and their intra-function features $I_{F_a}$, $I_{F_b}$, then the Euclidean distance between the two features is computes as follows:

$$D(I_{F_a}, I_{F_b}) = ||\delta(I_{F_a}, \theta) - \delta(I_{F_b}, \theta)|| \tag{3.1}$$

Where $\delta(I_F)$ represents an embedding produced by a CNN network for a specific intra-function feature. The objective of training phase is to find $\theta$ such that the above described distance is minimal in case the two functions are similar, and large otherwise. For this reason, the researchers aims to minimize the following loss function:

$$L(\theta) = Average_{(I_{F_a}, I_{F_b})}\{y \cdot D(I_{F_a}, I_{F_b}) + (1 - y) \cdot max(0, m - D(I_{F_a}, I_{F_b})\} \tag{3.2}$$

Where m is a pre-defined hyper-parameter. Therefore, the objective function of the training is:

$$argmin_{\theta} L(\theta) \tag{3.3}$$

Which is solved using Stochastic Gradient Descent and back propagation. The work also considers *inter-function features*, representing the caller-called relationships between functions. This feature is represented by the call graph of a program. For each function in the graph, researchers computes its inter-function embedding as a two dimensional

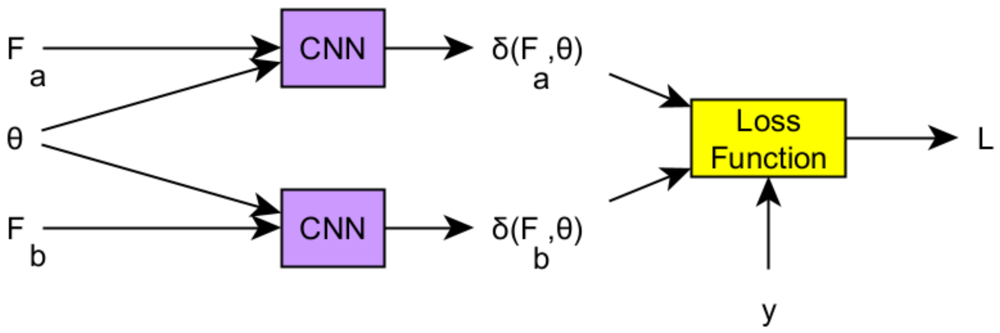

**Figure 3.4:** Siamese architecture used in [7]
during training of CNN model for intra-fuction feature extraction

vector $\gamma = (in_f, out_f)$, composed by the in-degree and out-degree of the function itself. Finally, the work evaluates *inter-module features*, consisting in the imported libraries from a specific function. Also in this case, feature are converted in an embedding vector to allow similarity comparisons.

These three feature types are used to evaluate the similarity between binary functions. The work first computes all three features for each of the two functions, then evaluates the distance between pairs of features. The similarity is therefore defined as the sum of these three distances.

Researchers define a dataset to evaluate their approach, composed by over 2'000'000 pairs of similar functions, compiled from two sources. The first is composed by 31 GitHub projects with over 9'000 different releases, compiled with GCC v5.4 and default optimization options. The second source includes binaries directly extracted from 895 packages of Ubuntu 12.04, 14.04 and 16.04, having over 1'800 different versions. To define ground truth for the dataset, researchers extracted the function name information using debug symbols.

Pei et al. [97] proposed TREX, a framework for training machine learning models for binary similarity and function search. The training approach is composed by two phases: pre-training and fine-tuning. During the pre-training, a model learns how to predict semantics of the execution traces when parts of the code are masked (e.g., values, registers, and opcodes). The model input is the sum of token embeddings of five sequences, representing features of the input function and its execution trace. The first sequence contains the assembly code of the forced execution path. Another sequence represents the numerical values of the each register before the execution. Three auxiliary sequences provide structural and syntactic information.

In the sequence containing numerical values each element is encoded using a bidirectional LSTM model into a fixed length sequence. The embeddings are computed as one-hot encoding in the other sequences. The pre-trained model is fine-tuned on the task of binary similarity between functions. The model is fed with pairs of function codes, that are embedded through a self attention layer, then fed into 2-layer perceptron. The fine-tuning tries to minimize the cosine embedding loss between the ground truth and the cosine distance between embeddings of the considered functions.

The model is trained and evaluated on a dataset composed by 13 open-source projects, obfuscated and compiled with GCC v7.5, 3 optimization levels and 4 architectures (x64, x86, ARM, MIPS).

In [78], Li et al. proposed an approach for computing binary similarity through graph matching networks. The approach is based on the evaluation of the similarity score between pairs of graphs. Each graph is represented as G=(V,E), where V and E are respectively the set of nodes and set of edges. Additionally, nodes in V and edges in E can be related to feature vectors representing for example the edge direction and node type. Nodes and edges without features have a default vector of ones.

To evaluate the similarity between graphs, researchers proposes two models, one based on graph neural networks (GNNs) and the other based on graph matching networks (GMNs). The GNN model is composed by three components: an encoder, a set of propagation layers, and an aggregator layer. The encoder embeds node and vector features into individual embeddings, using separate multi layer perceptrons. Then, propagation layers iteratively update the node embeddings in order to include information regarding their neighboring nodes. The aggregator takes as input the updated embeddings and transforms them into a vector representation of the input graph. These embedding vectors are then compared to evaluate the similarity between graphs. The comparison is performed using relevant metrics, such as for example the cosine similarity.

Researchers also proposed a GMN model, that allows to compare two graphs in input using a similarity score. The model considers simultaneously both graphs when updating node embeddings. The approach also uses a cross-graph attention mechanism, evaluating the matching between nodes of the two graphs in input.

The proposed models have been evaluated with various experiments, including the function search task. For this problem, researchers compiled multiple binary functions. These are obtained from the compilation of an open source package using GCC, Clang and different optimization levels. The training phase is performed on control flow graphs of binary functions. The trained models are then asked to search similar functions in a set of binaries.

An alternative approach is proposed by Massarelli et al. with SAFE [87], a model based on a self-attentive neural network for solving the task of binary similarity and function search. Researchers creates instruction2vec (i2v), an embedding model for encoding the input instructions. More precisely, the approach uses word2vec [90] to map disassembled instructions into numerical vectors and outputs a sequence of these embedding vectors.

Researchers trained two embedding models, one specifically designed for ARM architecture and the other one for AMD64. The embedded instructions are fed into a self-attentive neural network. The bi-directional RNN maps each embedding vector in a sequence of instructions into a summary embedding representing an entire function. These summary vectors are merged into a matrix H, where each row represents a function embedding. A two-layer attention mechanism is applied to assign weights on relevant function summaries, producing an attention matrix A. The resulting embedding matrix B is obtained as the product between A and H. The embedding matrix B is then transformed into a vector using a two-layer fully connected network, activated through a ReLU function.

The network has been trained using a Siamese architecture, composed by two identical bi-directional RNNs. The two networks takes as input a set of function pairs and the corresponding classification labels. For each pairs of functions, the architecture computes the cosine similarity between the output embeddings, assigning +1 to similar functions and -1 otherwise. The goal of the training is to minimize the difference between the predicted similarity score and the corresponding ground truth label. The model has been evaluated on several tasks, including binary similarity, function search, vulnerability search, semantic classification and APT classification. For each task, a specific dataset is used. For instance, in the function search experiment the model is trained on a dataset comprises binary functions compiled for AMD64 architecture. Then, the trained model is evaluated using binaries resulting from 12 compilers and 4 optimization levels.

Another significant contribute in this context is the one provided by Ding et al. with Asm2Vec [35], which uses neural networks to encode functions into vectors. The approach is based on PV-DM model, which learns the representation of a corpora from the corresponding text tokens. The model is adapted in Asm2Vec in order to represent assembly functions using their control flow graphs.

The model first retrieves disassembled functions and their control flow graphs using IDA Pro disassembler. Each function is represented as a set of sequences representing the potential execution paths of the control flow graph. For optimization purposes, Asm2Vec also introduces function inlining. This technique consists in replacing function calls with their code. After the inlining, Asm2Vec generates the sequence of execution paths. The approach randomly selects all edges from the control flow graph and generates the set of execution paths. In addition, the model performs a set of random walks on the CFG, to prioritize specific code blocks that are frequently executed. The model has been evaluated on stripped binaries through various tasks, including function search and vulnerability search.

### 3.2.2 Function Naming

The task of function naming asks to infer an appropriate name for a given binary function. A recent series of papers is focused on the problem of function naming, especially on stripped binaries. Examples are punstrip [94], a tool based on a probabilistic model for retrieve function names, XFL [95], based on a multi-label learning approach, and NFRE, a framework that retrieves function names using CFG and instruction sequences of assembly functions [46].

However, the majority of these works are focused on deep neural networks. An example of this approach is the one proposed by Artuso et al. in [9]. The research compares performance of two deep neural networks, namely Seq2Seq [11] and Transformer [121] architectures. These models are fed with sequences of normalized assembly instructions. The normalization is performed by substituting special symbols to memory addresses and immediate values under a specific threshold value.

The Seq2Seq model is composed by an encoder and a decoder. The encoder component includes a bi-directional RNN and LSTM cells, and the decoder is composed by a RNN with an attention mechanism. The transformer model has an encoder-decoder architecture, based on attention mechanism. Researchers built and distribute a dataset for evaluating the two models on the function naming task. The dataset includes over 8'800'000 stripped functions, compiled from various packages, including C libraries, videogames, and networking applications. The approach shows that a model pre-trained on the proposed dataset achieves significant performance when fine-tuned for specific tasks, such as the prediction of function names over malware samples.

Function naming problem has also been evaluated in NERO [28]. This approach applies static analysis and neural networks to create augmented representations of call sites from stripped binaries. More specifically, the approach first constructs the CFG, indicating the call instructions. Then, NERO determines each call sites in the analyzed assembly procedure, and the expected number of arguments. To define the values to be attributed to each argument, NERO uses a technique called pointer-aware slicing. For each register storing a function argument, the slicing technique extracts the code sections corresponding to the calculation and storage of its values. The resulting information is used to augment the call sites. Therefore, augmented call sites are used to generate a CFG of the augmented call sites, representing the analyzed binary procedure. In this graph, nodes represent the augmented call sites, while edges indicate potential execution paths of the procedure. Researchers evaluate various neural architectures using this representation of binary procedures, including LSTM attention-based models, and transformers.

A different approach is introduced by DEBIN [55], a machine learning approach that predicts debug information from stripped binaries. The assembly code is first translated in BAP-IR. This is an intermediate representation, showing code semantics in an architecture independent way. From BAP-IR, the approach extracts known and unknown data. The first information is already present in the binary code (e.g., constants, instructions), while the latter is an information that must be retrieved because it has been removed during stripping (e.g., registers storing values of variables). DEBIN uses a binary classifier to recovery the unknown information, that is trained on pairs of registers and memory offsets. The corresponding ground truth labels indicate if the pairs represent variables.

Therefore, DEBIN builds a graph representing undirected dependencies among code elements of BAP-IR. On top of this graph DEBIN runs a probabilistic model, the conditional random field, to predict the missing debug information. The predicted function properties are converted into DWARF format and used to update the stripped binary that is then returned as output of the process.

### 3.2.3 Signature Recovery

The problem of signature recovery asks to predict the parameters of a provided binary function, indicating the corresponding number of arguments and types. This task has received less attention than other tasks on binary functions.

The reference paper in this case is EKLAVYA [22], an approach that recovers function signatures through neural networks. The model is composed by an embedding component and an argument recovery component. The embedding part constructs an embedding vector for each instruction through the skip-gram model. The distance between vector embeddings represents the similarity between the corresponding instructions. For instance, the cosine distance between *push %edi*, *pop %edi* has roughly the same value of the same distance computed on the pair *push %esi*, *pop %esi*.

To recover the function signatures, the argument recovery component trains different four RNNs, one for each sub-task of the signature recovery problem. The sub-tasks are the following: argument count based on caller's instructions, argument count based on callee's instructions, argument type recovery based on caller's instructions, and argument type recovery based on callee's instructions. The model has been evaluated on two datasets, compiled with GCC and Clang in 4 optimization levels and x64, x86 instruction sets. The first dataset includes 3 Linux packages, and more than 270'000 functions for both x86, x64 architectures. The second dataset includes 8 packages and more than 370'000 functions for x86 and x64 instruction sets.

A more general problem related to signature recovery is the decompilation of binaries. Many works studied the problem, with and without neural networks [44, 65, 66]. However, the task considered for our study is focused on retrieving only the signature from a binary function and not the entire source code. Another related problem is the one of finding low level patterns in binaries [39, 40]. These patterns can be used to support the recovery of the arguments and return types of a function.

### 3.2.4 Compiler Provenance

The task of compiler provenance requests to classify a given binary function with respect to the compiler and/or optimization level used to generate it. This task has been studied in several papers. For example, the model introduced in [106] represents a binary program as sequence of idiom features. This kind of representation removes details such as memory offsets and literals. The approach uses conditional random fields to model the likelihood of compiler labels on the binary code. The method has been evaluated on various experiments, including binaries from single and multiple source compilers.

In [105], instead, the task of compiler provenance is evaluated as a classification problem. The approach generates a CFG for each analyzed binary and automatically selects features of the code. Researchers propose a tool, called ORIGIN, that extracts features from binaries and classifies code with respect to its source toolchain. Binaries are parsed through an existing library to reconstruct the interprocedural CFG. The library considers each function as an intraprocedural sub-graph, allowing to derive important features from the code. The approach also extract idioms, short instruction sequences containing wildcards. The approach also defines graphlets, that are non-isomorphic sub-graphs of the CFG that indicate the relationships between code blocks. There exist two types of graphlets: summary graphlets, focused on which type of instructions are present, and branch graphlets, focused on control flow instructions. The approach also extract the starting address of each function. Each function is associated to a binary vector, representing the presence of each feature. The model is implemented using an SVM classifier.

Another approach is proposed by BinComp [102], that identifies the compiler of origin using stratified components. BinComp extracts a set of syntactic features and compiler tags from known compiled code. Therefore, it uses ACFGs to extract semantics features from binaries. The solution allows to identify the compiler provenance and the optimization level.

More precisely, BinComp first preprocesses the input code to replace values in the symbolic constants. The preprocessed source code is then compiled. The approach selects code features from the resulting assembly code and creates a mapping between assembly and high-level code functions. This association allows to understand the compiler operations and on the code. Then, the compiled is linked to the required libraries. The resulting binary file is therefore disassembled to retrieve tags generated by the compiler used.

In HIMALIA [20], neural networks are used to retrieve the optmization level of the input binary function. The architecture of HIMALIA is divided in three components, an embedding layer, two bi-directional GRU layers, and a classification layer including a 4-classification model and a 2-classification model. The embedding layer is used to represent an input binary as a list of functions. The embedding is performed using a trainable model. In this layer, each function is encoded into a vector by applying word embedding to the corresponding instructions. Then, two bi-directional GRU layers are applied to select semantic features of the input instructions.

Finally, the model retrieves compiler information using two classifiers. The 4-classifier model is used to detect the optimization level of the function within O0, O1, O2/O3, and Os. The 2-classifier refines the O2/O3 prediction, distinguishing between O2 and O3 classification label.

In [86], researchers proposed an approach for automatically extracting features from the CFG. The model uses a graph embedding neural network to create a vector representation of the CFG. The architecture is composed by a two components. The first component is the Vertex Features Extraction, associating each graph vertex to a vector representation. Researchers uses an unsupervised approach for extracting code features, showing its improvement with respect to manual techniques. Each instruction is embedded using i2v model through skip-gram technique. This approach infers the surrounding instructions by analyzing a given instruction. Therefore, vector embeddings are aggregated to create a feature vector for the vertex of the CFG, representing the functions in the code. The aggregation is computed with two different techniques, namely i2v_RNN and i2v_attention.

The second component of the architecture is a Structure2Vec model, generating the complete graph embedding from the aggregation of the vertex vectors. More sepcifically, each vertex in the CFG is assigned to a vector, that is dynamically updated during the training phase. The dynamic update is performed in rounds, where the information of neighboring vertices are included in the embedding. After this phase divided in rounds, a final graph embedding is computed. This is obtained as weighted sum of all vertex vectors.

The approach has been used to solve two tasks, namely binary similarity and compiler provenance. For each task a different dataset is used. In detail, for the task of binary similarity. researchers define a dataset of over 95'000 graphs. The dataset is derived from the compilation of two versions of OpenSSL with GCC v5.4 and 4 optimization levels. In additions, packages has been compiled for both x86 and ARM instructions sets.

For the compiler provenance task, researchers define two datasets. The first includes roughly 450'000 functions from various projects. The projects are compiled for AMD64 with GCC v3.4, GCC v5.0, and Clang v3.9 and 4 optimization levels. The second dataset is composed by over 1'500'000 functions from different projects, compiled with 11 compilers and 4 optimization levels.

# Chapter 4

# Benchmark for Binary Function Representations

Deep learning advancements are significantly impacting the research on binary code analysis, allowing researchers to develop various DNNs for solving binary tasks. These architectures focus on learning a general representation of executable code, making them applicable to widely different problems on binaries. However to ensure their generality, these DNN architectures have to be tested on several diverse tasks.

A valuable approach to comply with this need is the adoption of standardized benchmarks, as proposed by the NLP reasearch community. These benchmarks are datasets used to evaluate the performance of DNN architectures on different tasks. Well-known examples of this approach in NLP field are: *Stanford QUestion Answering Dataset* (SQUAD) [103], focused on reading comprehension tasks, *General Language Understanding Evaluation* (GLUE) [123], that is a collection of multiple datasets, *GLUE for Code-Switched Languages* (GLUECoS) [67], designed for evaluating models on code-switching text, and *Knowledge Intensive Language Tasks* (KILT) [98], focused on tasks requesting additional information. Each benchmark could also contain different types of tasks. As an example, the GLUE benchmark contains various tasks associated to different datasets, such as Corpus of Linguistic Acceptability (CoLA), Stanford Sentiment Treebank (SST-2), Recognizing Textual Entailment (RTE), and Quora Question Pairs (QQP). These multi-task benchmarks are designed to evaluate the generality of a new developed architecture in a wide range of challenges in NLP context.

However, the binary analysis community does not currently dispose of any common multi-task benchmarks. As illustrated in the literature, all researches in this field are primarily relying on their own custom datasets, which have been specifically designed for solving individual problems under analysis (see, as an example [87], [129], [35], [9], [86]). This approach, however, leads to two significant issues.

The first concern is the *incomparability of results* between different works. Performance metrics obtained could, and often do, change drastically depending on what dataset has been used. This makes it difficult, if not impossible, to decide which architecture has performed better for a given problem. The only way to guarantee fair comparison of results is to evaluate different candidate solutions on the same set of data. Unfortunately, making this evaluation is also not trivial. Datasets and prototype DNN models are not often publicly available. In this case, the replication of necessary data for a proper comparison could be an important effort.

The second problem is the resulting *limited generalizability* of the models. The current trend of research is to create ad-hoc DNN architectures for each problem on binary code. While effective for certain tasks, these models can not be reused in other contexts. For this reason, the creation of generalized DNN architectures could be beneficial for binary code analysis. In the NLP community, generalized solutions are defined as pipelines of general-purpose neural networks and are designed to learn a general representation of natural languages. The resulting DNNs are used to solve several different tasks, usually after being fine-tuned. The fine-tuning process essentially consists in taking a pre-trained architecture and training it on task-specific datasets, such as those used in benchmarks. This approach leverages the pre-trained knowledge of the model, to let it learn the new task faster and with better performance.

Therefore, the development of a specialized benchmark is necessary for evaluating DNN architectures on binary tasks. An ideal benchmark should comprehensively assess a candidate DNN on heterogeneous tasks. This includes problems related to code syntax, such as the identification of compiler type used to generate a given binary, as well as tasks related to code semantics, such as understanding the purpose of a function within larger binary code snippets. The adoption of common benchmarks in binary analysis will introduce a unified way for testing and evaluating neural network models designed for binary representation tasks. Furthermore, this initiative could foster the creation of general, multi-task DNNs architectures that hold similar significance in binary analysis as BERT does in the NLP community. This would represent a significant advance in the capabilities and efficiency of binary analysis performed with DNNs.

For these reasons, the first contribution of the thesis introduces a benchmark, called *BinBench*, including five different tasks for binary function representation. The dataset associated to the tasks comprises two parts: one containing the original binaries and the other consisting in JSON files representing each binary function. Upon these five tasks, we have also defined five challenges containing pre-defined predictions for training and testing new DNN architectures.

The benchmark has been presented in the paper *"BinBench: a benchmark for x64 portable operating system interface binary function representations"*, published on the journal PeerJ Computer Science in 2023. The dataset is publicly available on FigShare [27]. The online version of the benchmark is published on Eval.AI [26], an open-source platform for the evaluation and comparison of machine learning and artificial intelligence algorithms [130].

The chapter is divided in four sections. The section 4.1 gives an overview of different benchmarks available in literature. The section 4.2 describes the tasks included in the proposed benchmark. The section 4.3 describes the construction of the dataset associated to the benchmark. Finally, the section 4.4 evaluates different models on the tasks of BinBench.

## 4.1 Background on Benchmarks

This section analyzes various widely recognized NLP benchmarks, used to evaluate the overall quality of DNN architectures. For generality purposes, these benchmarks include multiple heterogeneous tasks on which the architectures can be tested and also compared with respect to other models. The section also provides a description of a benchmark called BinKit [69]. The available literature suggests that BinKit is the only benchmark specifically designed for binary code analysis. However, it currently focuses on a single task. To the best of our knowledge, there is no benchmark including multiple tasks for binary analysis besides BinBench.

### 4.1.1 ERASER

Evaluating Rationales And Simple English Reasoning (ERASER) benchmark [34] is specifically designed to test interpretable NLP models. This benchmark comprises a collection of datasets, each one divided in train, validation and test sets. Datapoints, information to be predicted for a certain tasks, are associated to three data: input information, human-annotated rationales and prediction labels. The rationale is textual evidence used in support for explaining the model's output, while the labels represents outcome expected for that prediction. ERASER utilizes existing NLP datasets, that are focused on different textual corpora. This allows to compare performance of models on standardized datasets that are already widely-accepted in the NLP community. The *Evidence Inference Dataset* [76], for example, contains texts related to randomized controlled trials. In this case, the task is to predict whether and how a given intervention has influenced the outcome. *BoolQ* [24] is, instead, composed by text snippets from Wikipedia, associated with questions to be answered with yes or no.

The ERASER benchmark evaluates not only the predictions obtained by a model, but also the plausibility of rationales it have extracted. To evaluate the quality of the provided rationale, researchers proposed two metrics: comprehensiveness and sufficiency. The first metric indicates whether the model has selected all features requested for the prediction. The second metric, instead evaluates if the extracted rationales are enough to make predictions.

### 4.1.2 GLUE

General Language Understanding Evaluation (GLUE) [123] is a benchmark for testing models on natural language understanding tasks, focused on English sentences. The benchmark uses already existing datasets, each one associated to a different challenge. GLUE includes classification tasks for single sentences and sentence pairs, and regression tasks. An example of single sentence classification task is *Corpus of Linguistic Acceptability* [126], where the model have to decide if the given word sequences are grammatically acceptable in English. Another example of classification task for single sentences is *Stanford Sentiment Treebank*, that allows to test sentiment predictions of models. *Recognizing Textual Entailment* (RTE), *Microsoft Research Paraphrase Corpus* (MPRPC) [36] and *Quora Question Pairs* (QQP) are, instead, tasks for sentence pairs classification. RTE is a task for evaluating semantic relationship between sentence pairs, having different vocabulary and syntax. MRPC and QQP instead require to determine the semantic equivalence between sentences. The benchmark also includes inference tasks, such as *Multi-Genre Natural Language Inference Corpus* (MNLI) [127]. This is composed by a set of sentence pairs, premise and hypothesis, on which a language model has to understand the type of entailment between the two sentences.

#### GLUECoS

GLUE for Code-Switched Languages [67] is a benchmark for evaluating the generalization of multi-language models on tasks related to code-switched data, in English-Hindi and English-Spanish languages. Starting from the idea of GLUE, researchers created a new benchmark including five tasks of GLUE and a new task, called *Natural language inference for code-switching.* In this task is asked to infer a positive or negative relationship between pair of sentences. Another interesting task is *Named Entity Recognition* (NER), consisting in classifying named entities (e.g., organization, location, etc.) in a given text snippet. Two corpora are provided for NER, one for English-Hindi and the other for English-Spanish.

### 4.1.3 KILT

Knowledge Intensive Language Tasks [98] is a benchmark for evaluating general purpose models on knowledge-intensive language tasks. The benchmark can be used to test any type of model, as it simply requires textual outcomes without specifying any details regarding architectures involved in the inference.

KILT is a collection of tasks developed on various datasets. Researchers also use a single Wikipedia snapshot to support the information provided by the datasets. The snapshot is shared among all the tasks of KILT.

Using the same knowledge base for all tasks simplifies the comparison between models. This also reduces the need to re-train models for each task performed. Researchers provided not only datasets for the tasks, but also a complete interface for evaluating the responses provided by the models.

### Tasks

KILT defines five different types of tasks: Fact Checking, Open Domain Question Answering, Slot Filling, Entity Linking, Dialogue.

The first task, Fact Checking, is a classification problem. It requires to verify a claim using the available evidences. The task is based on FEVER [119], that is a dataset composed by claims generated from Wikipedia's sentences. In the task of Open Domain Question Answering, the model is given a question and it has to find the answer in the knowledge source. The model's output also includes the location of the information used for producing the answer. The Entity Linking task, instead, asks to associate entities to the Wikipedia pages where they have been mentioned. The outputs required for the task are titles and unique identifiers of Wikipedia pages. To solve Slot Filling, the model has to find information regarding the relationship between entities in a large text corpora. Finally, in the Dialogue task, the model needs to have a conversation on with the user. The dataset used for the task is grounded with the knowledge source of Wikipedia.

### KILT Challenge on EvalAI

Researchers published the KILT benchmark on EvalAI, that is a platform where machine learning models can assess their performance on different challenges. For the KILT challenge, users upload individually the predictions generated by their models for each task.

KILT benchmark provides users with a JSON file for each task, containing data-points to be predicted for each task. Users fill these JSON files with models' predictions and then upload them back to EvalAI platform. Therefore, the KILT's script analyzes the predictions and computes specific metrics on them. These metrics are then published on the public leaderboard of KILT. In this way, model's performance can be compared against others, including the baseline models established for each task.

The advantage of using EvalAI is that users can easily evaluate their models for different tasks, by just uploading the list of predictions in the platform. Also, this approach facilitates the comparison of different models, including baselines provided by the researchers. For this reason, we have opted for the same approach for the benchmark we proposed. We have published BinBench on EvalAI, including all the tasks and the corresponding metrics.

### 4.1.4 BinKit

Kim et al. propose a benchmark, known as BinKit [69], focused on the evaluating models on binary similarity between code snippets. The benchmark is associated to a public dataset, and an interpretable model for evaluating the proposed approach. To the extent of our knowledge, in addition to BinBench, BinKit is the only benchmark available for the field of binary code analysis. Moreover, BinKit is limited to a single task is insufficient, making it inadequate for evaluating the generalizability of models.

#### Binary Similarity

Researchers define four steps of analysis for binary similarity: syntactic, structural, semantic, vectorization, and comparison.

*Syntactic analysis*, consisting in extracting the Abstract Syntax Tree (AST) or disassembled of the code, is used to understand the syntax behind input code snipped.

*Structural Analysis*, instead, is focused on retrieving control flow graphs (CFGs) and call graphs (CGs) of the binary code. In detail, CFG is a directed graph, that uses arrows to show all possible paths of execution for a function or a program. It consists in two main parts. The first component is the set of nodes, representing basic blocks of code. A basic block is composed by a set of consecutive instructions ending with jump or return statement. The second component, instead, is the set of edges representing jumps from current basic block of instructions to its successor. In other words edges represents the program flow from one block to another. CG, instead, is a directed graph representing function calls within a program. Each node in CG represent a function, while edges indicate which function called another one.

Using the information of Structural Analysis, it is also possible to perform *Semantic Analysis* of a program. This step consists in generating features capturing the code's semantic.

The subsequent step is the *vectorization* of all the information from previous analyses to obtain feature vectors. Finally, the approach computes the similarity *comparison* between vector features. The output of this step is a value between 0 and 1, representing the similarity level of two functions.

#### Features

The study classify vector features in two categories: *presemantic* and *semantic*. Presemantic features represents attributes of binary code obtained from syntactic or structural analysis. Researchers consider two types of attributes: non-numeric, consisting in program properties, and numeric, indicating the number of occurrences for specific properties in the code. Semantic features, instead, are those derived from semantic analysis. These features can be extracted through techniques like symbolic execution or machine learning embedding vectors.

### Dataset Creation and Evaluation

The dataset includes more than 240'000 binaries, obtained from the compilation of 51 GNU software packages over 8 different architectures. The compilation is performed with 9 compiler versions (GCC 4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0, and Clang 4.0, 5.0, 6.0, 7.0) and 5 optimization levels (O0, O1, O2, O3, Os). The dataset has been sanitized, removing invalid binaries, and streamlined from duplicate functions.

The benchmark also includes ground truth information. This associates a label to each function in the dataset, containing function name, source package, name of the binary file, name of the corresponding source file, and reference line numbers.

Researchers implemented also an interpretable model that evaluates binary similarity between functions. To compute similarity, the model uses an approach based on manually selected numeric presemantic features. The binary similarity is computed as a scoring metric using the average of relative differences between all selected features.

Given a numeric feature f, where $A_f$ and $B_f$ are its values for functions A and B. The relative difference D between the two values is:

$$D(A_f, B_f) = \frac{|A_f - B_f|}{|max(A_f, B_f)|} \tag{4.1}$$

Considering N features in the feature set, the similarity score s is computes as follows:

$$S(A, B) = 1 - \frac{(D(A_1, B_1) + D(A_2, B_2) + ... + D(A_N, B_N))}{N} \tag{4.2}$$

## 4.2 Task

This section provides a description of the tasks included in BinBench, the benchmark we propose as first contribution of the thesis. The tasks are the following: *binary similarity, function search, function naming, compiler provenance,* and *signature recovery.* These tasks capture distinct characteristics of binary functions and have received significant attention within the relevant literature, as also showed in the chapter 3.

The selection of known tasks for BinBench was driven by several reasons. Firstly, we want to include tasks that have demonstrable real-world relevance. This ensure that the benchmark can be used to assess the practical usability of DNN architectures within relevant scenarios. Furthermore, we opted for well-known tasks because they have already been accepted within the research community. This choice could encourage the adoption of the proposed benchmark by the research community. Finally, the adoption of known tasks facilitates the identification of existing models that can be used as baselines for the benchmark. Baseline scores represent references for evaluating the performance of new models on each task. The comparison with baseline scores allow researchers to understand whether a model has better performance with respect to the existing solutions.

The tasks of BinBench can be categorized in two groups: semantic tasks and syntactic tasks. The semantic tasks require the evaluated architecture to capture the semantic aspects of binary functions, abstracting from the syntactic representation of the code. The tasks that are part of this category are binary similarity, function search and function naming. For the binary function and function search tasks, a model has to understand the semantic of binaries to recognize function pairs compiled from the same source code. For the function naming task, a model needs to develop a deep understanding in order to assign a meaningful name to a code snippet representing a binary function. The syntactic tasks, on the other hand, prioritize the analysis of the syntactic aspects of binary functions. This category includes the compiler provenance task and the signature recovery task. The first requests to use syntactic information to infer the compiler having generated a specific binary function. The latter task requires to understand code syntax in order to determine the number and type of arguments taken as input by a binary function.

In the following we provide a detailed examination of the tasks included in the benchmark. In particular, for each task we provide a comprehensive explanation of its purpose, its formal definition, and a discussion of the metrics used to evaluate models specifically on that task.

### 4.2.1 Binary Similarity Task

The task of binary similarity asks the network to understand whether two binary functions are based on the same source code but generated using different compilers and/or optimization levels. In this task, a network is required to abstract from the syntactic difference created by the compiler. For this reason, the binary similarity is considered a semantic task.

Assessing the similarity between binary functions is a challenging problem. This complexity is caused by the combination of compiler and optimization level used. These factors can introduce substantial variations in the resulting binaries, even when compiled from identical source code [129]. The problem has been extensively investigated in the last years [54]. The task of binary similarity has practical uses in various fields, such as clone search, copyright infringement dispute, etc.

For our benchmark, we opted for the definition of binary similarity introduced in [35] and subsequently also used in [87]. This approach expresses the problem as a comparison between pairs of functions.

Two binary functions $f_1, f_2$ are considered *similar*, $f_1 \sim f_2$, if they result from the compilation of the same original source code $s$ with different compilers. Essentially, a compiler $c$ is a deterministic transformation mapping a source code $s$ to a corresponding binary function $f^s$. We considered as a compiler the specific software used for the compilation, e.g. gcc-5.4.0, together with the parameters influencing the compiling process, e.g. the optimization flags -O[0, ..., 3].

For this task we provide a set of unlabeled binary function pairs $p_1, \ldots, p_n$ to be evaluated. The output requested is the classification of each function pair $p_i$ as similar (label $= +1$) or dissimilar (label $= -1$). Examples of binary similarity are showed in figure 4.1.

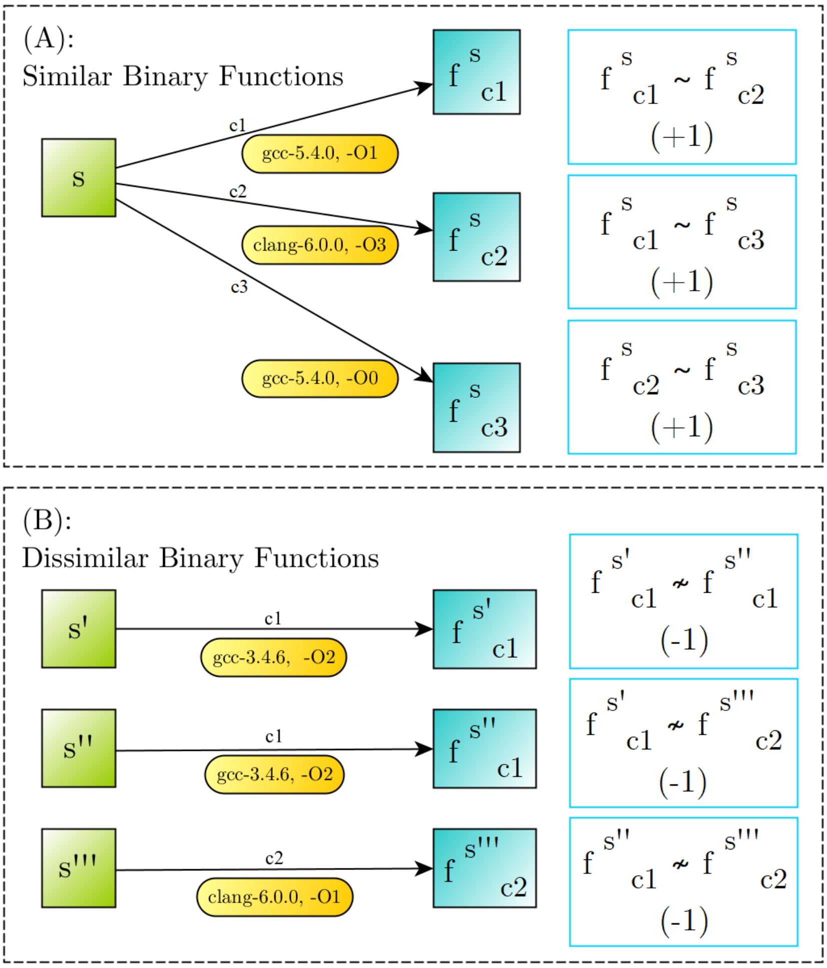

**Figure 4.1:** Examples of binary similarity. (A) Similar binary functions are generated by compiling the same source code. (B) Dissimilar binary functions are obtained from the compilation of different source code.

### Metrics

For binary similarity, the benchmark assesses the quality of a model's solution using the Area Under the Curve (AUC), considering the classification task as explained above. The AUC is the area under the Receiver Operator Characteristic (ROC) curve, which represents the relation between True Positive Rate (TPR) and False Positive Rate (FPR). Therefore, the AUC is used to show the overall classification capabilities of an observed model [62].

### 4.2.2 Function Search Task

The task of function search is similar to the binary similarity one. For both tasks, the goal is to identify functions that are considered similar according to the definition explained previously.

The main difference is that the two problems belong to different task categories. The binary similarity task is considered a *classification task*, because it requests to classify pairs of functions as similar or dissimilar. The function search task, on the other hand, is a *retrieval task*. Given a *query* function, the task requests to find the corresponding $K = 20$ most similar functions in a *database*. The value assigned to the parameter $K$ has been defined in relation to $S$, the average number of similar functions included in the entire dataset. The value of $S$ is slightly greater than $K$. As for the binary similarity task, even the function search is considered a semantic task.

This task reflects a real-world application of the binary similarity search where a specific sensitive function has to be searched on a database. A typical example is the search of malware code snippet or part of copyrighted software.

We formally define the function search task as follows. Given a database $DB$ represented as a set of $n$ distinct binary functions, and a set of queries $Q$ containing q binary functions $\{f_1, f_2, \ldots, f_q\}$. For each function $f_i \in Q$, the function search task requests to return a set $A_i \subset DB$ of size $K$. The functions in $A_i$ have to be similar to function $f_i$.

#### Metrics

The benchmark evaluates the quality of a model's solution using precision, recall, F1-score and normalized discounted cumulative gain (nDCG). The evaluation is computed on the $K = 20$ most similar functions proposed in the solution.

More precisely, the *precision* represents the fraction of correctly classified samples over the total retrieved samples. Therefore, given the true positives (TP) and the false positives (FP), the precision is computed as follows.

$$precision = \frac{TP}{TP + FP} \tag{4.3}$$

The *recall*, instead, measures the number of correctly retrieved instances over the total number of instances of the correct answer.

$$recall = \frac{TP}{TP + FN} \tag{4.4}$$

Where FN represents the false negatives (FN).

The *F1-score* shows the overall precision of a model and it is computed as the harmonic mean of precision and recall.

$$F1 - score = 2 * \frac{precision * recall}{precision + recall} \tag{4.5}$$

Finally, the *nDCG* measures the ranking quality for retrieval tasks. Given a binary function (i.e., the query), our goal is to retrieve its $K$ most similar functions from a database. The solution should place the effective similar functions in first positions of the list of returned functions.

As an example, consider the optimal query answering $s_{optimal} = (f_1, f_2, f_3)$, where each $f_i$ $_{(i=1,...,3)}$ is similar to the query $f_q$. Then, suppose to have two models $m_1, m_2$ that return the following solutions: $s_{m_1} = (f_4, f_1, f_2)$ and $s_{m_2} = (f_1, f_4, f_2)$, where $f_4$ is not similar to $f_q$. Both solutions include the same set of functions. However, according to the definition of nDCG, the solution $s_{m_2}$ is better that the solution $s_{m_1}$. This because $m_1$ places a non similar function at the beginning of the list, while $m_2$ places the same function in the second position. Formally, the nDCG is defined as follows.

$$nDCG(R_{\vec{f}}) = \frac{\sum_{k=1}^{k} \frac{similar(r_i, \vec{f})}{log(1+i)}}{idealDCG_k} \tag{4.6}$$

Where:
- $f$ is the query function
- $R_{\vec{f}} = (r_1, r_2, ..., r_k)$ are the top-k similar functions
- $similar(r_i, f)$ is equal to 1 if $r_i$ is similar to $f$, 0 otherwise
- $idealDCG_k$ is the discounted cumulative gain of the optimal query answering

The nDCG ranges between 0 and 1, and it depends on the *order* assigned to the returned similar functions. For this reason, the nDCG is greater when the similar functions are placed in first positions of the model's solution.

Consider the example discussed above. The optimal query answering is $s_{optimal} = (1, 1, 1)$, while two model solutions are $s_{m_1} = (0, 1, 1)$, $s_{m2} = (1, 0, 1)$. Each element in the solutions is 1 if the returned function is similar to the query, and 0 otherwise. Therefore, $s_{m_2}$ is a better solution than $s_{m_1}$, and $nDCG_{s_{m_2}}$ is greater than $nDCG_{s_{m_1}}$.

### 4.2.3 Compiler Provenance Task

The compiler provenance task requests to classify a given binary function with respect to the compiler and/or optimization level that generated it. The compiler provenance is categorized as a syntactic task. Models trained for this task learn to identify the syntactic structure generated by a compiler, without evaluating the semantic of the code.

The problem of compiler provenance was presented for the first time in [106]. This task has been further studied in several works [20, 86, 102, 105].

The identification of compiler provenance has practical applications in diverse real-world scenarios. For example, specific library detection toolkits such as IDA FLIRT [1] require knowledge of the compiler used to generate a given binary. Another application is the identification of program authors [17]. The compiler of provenance for an executable code is a necessary information for understanding who is the programmer that has compiled a particular binary code snippet.

To formally specify the task of compiler provenance, we first define the concept of compiler families. A family of compilers is a set containing all versions and optimization levels of a specific compiler (e.g., the Clang family includes the versions of Clang with all possible optimization levels). Using this notation, we can partition the set of possible compilers of origin, denoted as $C = \{c_1, c_2, \dots, c_k\}$, into compiler families $F = \{f_1, f_2, \dots, f_m\}$.

Therefore, the compiler provenance task requests to classify each given binary function $b_1, \dots, b_n$ with respect to the corresponding compiler family. In other words, for every function $b_i$ compiled with the compiler $c_j$ and any optimization level, the model is required to output the compiler family $f_v$ such that $c_j \in f_v$.

#### Metrics

We evaluate the Compiler Provenance task using four metrics: accuracy, precision, recall, and F1-score. The *accuracy* of a model is computed as follows.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.7}$$

Where TP are the true positives, TN are the true negatives, FP are the false positives, and FN are the false negatives.

### 4.2.4 Function Naming Task

The purpose of function naming task is to predict an appropriate name for a given binary function. This problem was first proposed in [55], and then further studied in various works [9, 28].

The function naming task asks a model to learn the semantic features of a binary function. These features are translated into a natural language description of the code, constituting a name. For this reason, function naming is considered a semantic task. A major challenge in evaluating function naming solutions is the wide variation in naming conventions that could be used by different programmers. An approach proposed in the literature addresses this issue by splitting function names into substrings [95]. The approach is detailed in the following metrics section.

[1] https://www.hex-rays.com/products/ida/tech/flirt/in depth.shtml

The task of function naming has practical importance in reverse engineering. Knowing the name of an unknown function can help an engineer understand the purpose of that code. This task can be applied, for example, to encryption or networking functions.

Formally, the function naming task is defined as follows. Given a vocabulary $V$ and a set of functions $f_1, f_2, \ldots, f_n$, the function naming task requests to assign each function $f_j$ to a string $s_j$ composed by words in the vocabulary $V$. The string $s_j$, representing the function name, has to be *"meaningful"*. Therefore, $s_j$ has to effectively capture the semantics of the given function and its role within the software [9].

### Metrics

The benchmark measures the quality of a function naming solution with several metrics: precision, recall, F1-score, and BLEU.

To be evaluated, each function name is represented as a list of tokens. More precisely, function names are first split on underscores. Then, English word segmentation [2] is applied on the resulting strings. This tool splits strings into "meaningful" English words. For instance, consider the function names `setValue`, `set_value`, and `setvalue`. These names have different naming conventions, while containing the same English words. Consequently, splitting these function names result in the same output: `["set", "value"]`.

In our benchmark we provide a dataset for training and evaluating models on every task. The dataset is divided in labeled split, used for training, and unlabeled split, used for testing (more details are given in the section 4.3). For the function naming task, the dataset's labeled split contains two labels for each sample. These labels represent the original function name, and the one resulting from the splitting technique. We believe that having both labels can be helpful to train models. For the unlabeled dataset, instead, an architecture has to predict only the function name composed by tokens of the vocabulary[3] used in the split procedure (i.e., tokens are the predicted labels). This list of tokens will be compared with the list of correct labels, obtained by applying the split procedure to the original function name. More precisely, for each function name we consider the following information.

- Two list of tokens: correct labels $l$ and predicted labels $p$,
- A function $x_a$, such that $x_a = 1$ if $p_a \in l$ and 0 otherwise ($\forall 0 < a \leq |p|$)
- The score of a prediction $score_p = \sum_{k=1}^{|p|} x_a$

[2] https://grantjenks.com/docs/wordsegment/

[3] https://github.com/grantjenks/python-wordsegment/tree/master/wordsegment

For each prediction, the benchmark computes the following *individual metrics*:

$$precision = \frac{score_p}{|p|} \tag{4.8}$$

$$recall = \frac{score_p}{|l|} \tag{4.9}$$

$$F1\_Score = \frac{2(precision * recall)}{precision + recall} \tag{4.10}$$

In addition, each prediction is also evaluated with the bilingual evaluation understudy (BLEU) score [92], measuring the quality of machine translations. This metric is language independent and returns a value between 0 and 1. Formally, BLEU score is defined as follows. Given the length of a prediction $c$, the effective length $r$, and the brevity penalty (BP)

$$BP = \begin{cases} 1 & \text{(if c >r)} \\ e^{(1-r/c)} & \text{(if c < = r)} \end{cases} \tag{4.11}$$

The BLEU score is:

$$BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n) \tag{4.12}$$

Where $N$ is the maximum n-gram length, $p_n$ is the modified precision score, and $w_n$ represents the corresponding weight.

For each solution proposed, the benchmark outputs a set of *returned metrics.* They are computed as average of each individual measure. Therefore, the returned metrics indicate the model's performance over all the predictions generated.

### 4.2.5 Signature Recovery Task

The signature recovery tasks consists in predicting the parameters of a given binary function. These parameters are the number and type of arguments the function takes as input. The signature recovery was proposed for the first time in [22], and has been solved as a sub-problem, also in [55]. As suggested by [22], the signature recovery task is useful for control-flow hardening [132] and taint-tracking [110].

The signature recovery is a syntactic task, because requires to an evaluated network to learn how the compiler handles parameters in the prologue of a function. In our benchmark, we selected the data types used by [22] as possible parameter types. In addition, we also included void, that should be interpreted as absence of parameters.

Formally, we consider a set of types $T$ : $\{\mathsf{pointer}, \mathsf{enum}, \mathsf{struct}, \mathsf{char}, \mathsf{int}, \mathsf{float}, \mathsf{union}, \mathsf{void}\}$ and a set of functions $f_1, f_2, \ldots, f_n$. Given a function $f_j$, the signature recovery task asks the network to output a multiset $P_j$ with elements in $T$. This multiset represents all parameters the function takes as input with their types. The prediction is a multiset that model functions having multiple parameters of the same type. For instance, when a function takes as input two integers, the correct prediction is $\{\mathsf{int}, \mathsf{int}\}$.

To prevent misinterpretation of data types by the decompiler, we opted for using only a subset of the data types available. We believe that this subset is sufficiently representative, as it is composed of the most diverse data types. More precisely, we have included int, because it can be used to represent both integers and boolean values (as 0 and 1). We have inserted pointer, that is used to perform memory accesses. We have also selected char, float and void data types to represent, respectively, non numeric symbols, floating numbers and absence of parameters. Finally, we have included enum, struct and union to represent complex data types.

### Metrics

The benchmark evaluates solutions for the signature recovery task using the accuracy, precision, and recall. These metrics are computed using the micro averaging method, which allows to properly evaluate a multi-class classification problem in imbalanced datasets. This method, however, produces metrics with same values [58].

## 4.3 Dataset

The proposed benchmark is associated to a dataset of binaries. These are extracted from 131 core packages within the Arch Linux repository [4], a Linux distribution optimized for x86-64 architectures. Each package is compiled using the tool *makepkg*, included in the Arch Linux package manager. The tool automates the building process of packages, by executing the instructions provided in the corresponding *PKGBUILD* shell script.

Each package has been compiled using GCC and Clang, two commonly used compilers for research purposes. To balance the total number of resulting binaries, we have considered five different versions for both compilers. For GCC we selected the versions $6, 7, 8, 9, 10$, while for Clang we used the versions $4, 6, 8, 10, 11$.

During the compilation, we keep debug symbols (-g) and compile the packages with four different optimization levels: -O0, -O1, -O2, -O3. We discard packages in two cases: when the compilation process fails, and when the binaries generated by different optimization levels are equal. The latter case is excluded to avoid the introduction of duplicated functions in the dataset. Duplicate removal is performed to prevent potential biases during evaluation. This process is also applied by various works in literature [22, 28, 87].

After the compilation phase, each binary function is translated into textual data and stored in JSON format. These JSON files serve as the input for models to address the tasks we proposed. For the test data split, we provide only these JSON files. For the training data split, we provide both the JSON files and corresponding binaries. The compiled files

[4] https://archlinux.org/packages/

could in fact be useful in case of custom training. It should be noted, however, that the provided binaries are not stripped of debug symbols. Therefore, this debug information might unintentionally influence models that are not supposed to use it. For this reason, it is recommended to strip binary files before using them as model inputs.

### 4.3.1 Implementation

To compile the selected packages, we set up Docker containers of Arch Linux and use parallel processing. The compilation process generates a total of 1'127'479 binary files. Arch Linux is a from-scratch distribution that can be configured to compile packages using the preferred compiler and optimization level. Its integrated package manager, pacman, takes care of packet dependencies.

From each compiled file, we extract a set of functions to be included in the BinBench benchmark. The disassembled codes and CFGs are retrieved using the Ghidra tool [5].

We remark that we do not use a symbolic or a dynamic dissasemble approach. Therefore, we do not predict the target of indirect jumps and calls. This approach is also used in [35, 87, 129]. We use $pyelftools$ APIs to extract information regarding function signatures [6]. The resulting textual data are stored in individual JSON files, one for each binary function. The total number of source code functions is roughly 132'000, that are compiled into 4'408'191 different binary functions. The average number of similar functions is $S = 26$.

We split the binary functions in two sets, namely the *labeled dataset* and *blind dataset*. The former is intended for the training process, while the latter is meant for the testing phase. The latter contains a subset of all compiled packages, that has been manually selected. This selection of packages includes different application types, in order to capture the majority of behaviours exhibited within software systems. For example, the blind dataset includes packages for network communication (e.g., OpenSSL), database management (e.g., SQLite), shell scripting (e.g., Bash) and archive management (e.g., Tar).

As mentioned earlier, the labeled dataset has two components. The first is the *binary component*, that is composed by the compiled files from which binary functions are extracted. The second is the *JSON component*, that is the set of extracted binary functions in JSON format. In the JSON component, each file represents a function and it is named with the corresponding function name. Each JSON contains the following fields:

- *asm*, instructions composing the function represented in assembly and bytecode formats. Each instruction block is linked to the corresponding offset;
- *called*, functions that are called within the code;
- *callers*, functions that call the function under analysis;

[5] https://github.com/NationalSecurityAgency/ghidra

[6] https://github.com/eliben/pyelftools

- *cfg*, control flow graph of the function. It is a directed graph, represented as a list of edges. The CFG is defined by two components: the pair of source and destination blocks, and the list of blocks composing the graph. Each block of the list is represented by the list of assembly instructions and the corresponding bytecode.
- *compiler*, compiler version used to produce the binary;
- *name*, function name retrieved using the debug information;
- *opt_ level*, optimization level used during the compilation;
- *origin_file*, name of the binary file from which the function has been extracted;
- *package*, name of the package containing the origin_file;
- *parameters*, list of function's parameters. Each parameter is represented with its name and type. Types can be one of the following: pointer, float, char, int, enum, struct, union, void (when the function takes no parameters);
- *return_ type*, type of the value returned by the function.

The two components of the labeled dataset are organized in mirrored structures and stored separate folders. In particular, packages are represented as parent folders, named with the package's name. Furthermore, each package contains several subfolders, one for every compiler version. Every compiler folder contains a set of subfolders, one for each optimization level. Finally, the optimization level folders are used to store the actual dataset files. More precisely, the JSON component stores functions in JSON format, named with corresponding function names. The binary component, on the other hand, stores binary files, named with the names of compiled files. Figure 4.2 shows the structure of a package folder included in the labeled dataset.

The blind dataset has a different structure with respect to the labeled one. This dataset is used for testing DNN architectures and evaluate their solutions through the online or offline evaluation script. For this reason, the blind dataset includes a set JSONs where each file contains only a subset of the fields given in the training set. This set of information is strictly required for solving the benchmark tasks. The fields included in these JSON files are the following: *asm*, *called*, *callers*, and *cfg*. The corresponding binary files are not provided for this dataset.

In addition, any duplicate functions are filtered out from the blind dataset. A function is considered a duplicate if it contains the same sequence of instructions of another function, without considering memory offsets and immediate values.

Finally, we discarded the folder structure representing packages, compiler versions, and optimization levels. Therefore, the JSON files are stored in a single "blind" folder, where each file is named with a unique ID (in replacement of the corresponding function name).

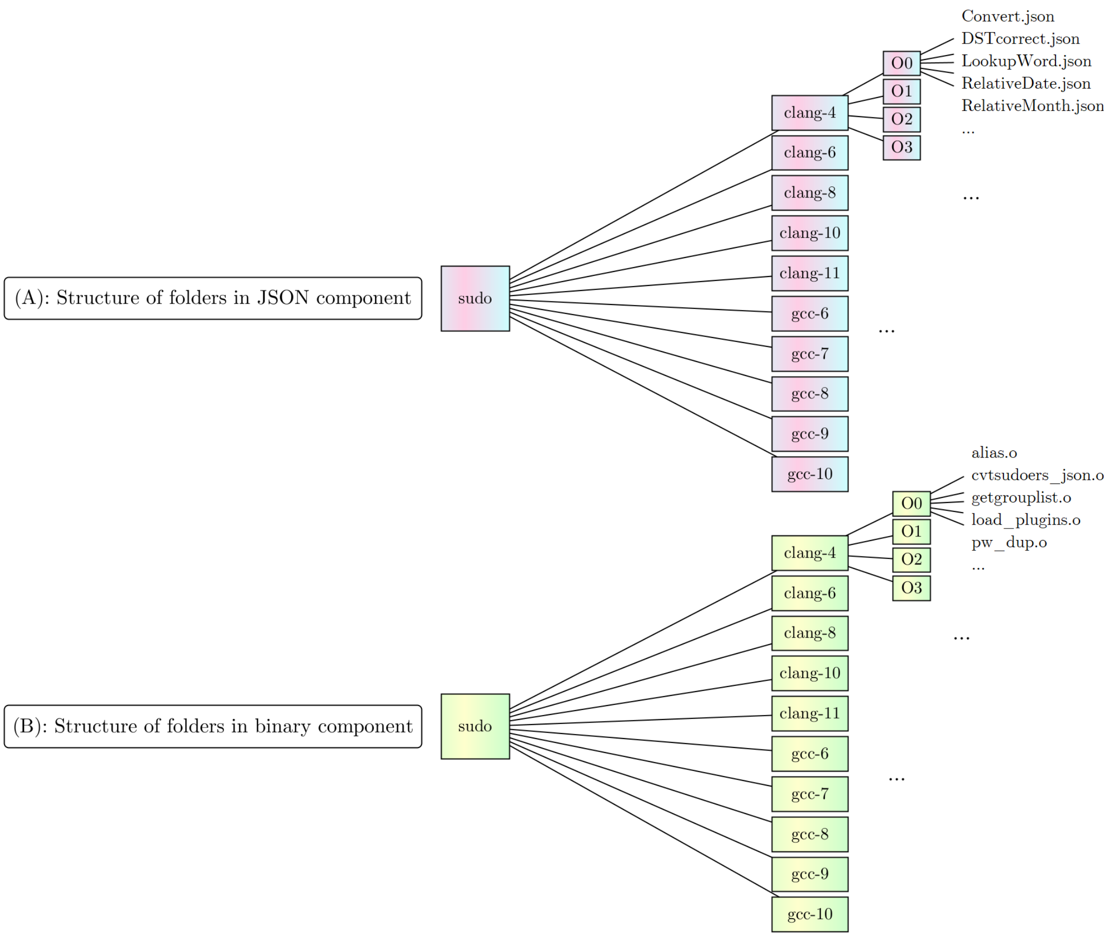

**Figure 4.2:** Folder structure of a package in the labeled dataset. (A) Folder in the JSON component. (B) Folder in the binary component.

### Task Datapoints

For each task we selected a set of datapoints, that represents the predictions to be computed by a model. A datapoint is a couple containing an entry in the dataset and the corresponding label to be predicted, that depends on the specific task.

Table 4.1 shows the total datapoints that has to be evaluated for each task. Table 4.2, instead, indicates the number of datapoints of the function search task. In this case, the datapoints are split in two sets. The first set contains the queries, that are the functions for which the model has to find their $K$ similars. The second set is the database, in which the $K$ similar functions have to be found.

For each task, datapoints are stored in individual JSONL files. In the labeled dataset, we provide individual JSONL files containing the task solutions (i.e., label fields of the datapoints are filled). In the blind dataset, instead, we provide two JSONLs for each task. In the first file, the label of datapoints are empty. In the second file, instead, the label fields are filled. These two files are used to evaluate the model performance as described in the following section.

| Task Name | Labeled Dataset | Blind Dataset |
|---|---|---|
| Binary Similarity | 732’376 | 243’044 |
| Compiler Provenance | 89’744 | 9’600 |
| Function Naming | 120’640 | 9’600 |
| Signature Recovery | 120’259 | 9’086 |

**Table 4.1:** Datapoints contained in each dataset split.

| Dataset | Query | Database |
|---|---|---|
| Labeled | 30’000 | 600’000 |
| Blind | 10’000 | 200’000 |

**Table 4.2:** Number of datapoints for the function search task.

We point out that these datapoints refer to a subset of the functions included in the entire dataset. However, each function in the benchmark can be used to train/test an architecture on the proposed tasks. Every JSON file, in fact, contains all the required fields for solving the every task of the benchmark.

### Task Evaluation

For each task, we request to infer a set of datapoints defined on the blind dataset. We provide two evaluation methods for each task: offline and online. The offline evaluation is computed through a script included in the dataset. The online evaluation, on the other hand, is hosted on EvalAI [7]. This is an open-source platform for testing the performance of machine learning algorithms. In EvalAI, each user can upload his own challenge and share it with the community. Each challenge can be divided in multiple phases, that represents different tasks to be solved in a benchmark. These functionalities are used for hosting our benchmark. We created a challenge named BinBench[8], including all the tasks defined in previous sections.

The online and offline scripts compare the predictions against the actual solutions. As explained previously, we provide two versions of the files containing the datapoints of the blind dataset. The first version represents the ground truth that is compared with the model’s solution. It contains the labeled version of the task datapoints. The second file is composed by unlabeled datapoints, and it is intended for entering model predictions. The labels inserted in this file are evaluated either offline, using the specific script, or online. In the latter case, the prediction file is uploaded on the EvalAI platform and evaluated using our online script. Depending on the task being solved, different metrics will be returned by the offline and online scripts. These metrics allow to verify the performance of a DNN architecture, comparing it with other models on various tasks.

[7] https://eval.ai/

[8] https://eval.ai/web/challenges/challenge-page/1980/overview

### 4.3.2 Comparison with other datasets

We compare our benchmark with the datasets available literature. For a fair comparison, we focused our comparison on the number of functions relevant to the BinBench tasks. Therefore, we have excluded portion of datasets that addressed tasks not included in BinBench. The results of our comparison are reported in Table 4.3.

| **Dataset** | **Number of Functions** | **Evaluated Task** | Open Source |
|---|---|---|---|
| BinKit [69] | 75’230’573 | ① | Yes |
| jTrans [124] | 25’877’011 | ①, ② | Yes |
| In Nomine Function [9] | 8’861’407 | ④ | Yes |
| $\alpha$Diff [81] | 4’979’586 | ① | Yes |
| **BinBench** | 4’408’191 | ①, ②, ③, ④, ⑤ | Yes |
| SAFE [87] | 548’133 ①, 581’640 ②, 1’587’648 ③ | ①, ②, ③ | Yes |
| Graph Embedding NNs [86] | 95’535 ①, 2’040’246 ③ | ①, ③ | Yes |
| TREX [97] | 1’472’066 | ①, ② | Yes |
| Toolchain Provenance [105] | 955’000 | ③ | No |
| BinFinder [100] | 783’767 | ② | No |
| Asm2Vec [35] | 139’936 | ② | No |
| Gemini [129] | 129’365 | ① | No |
| Eklavya [22] | 119’352 | ⑤ | Yes |
| NERO [28] | 67’246 | ④ | Yes |
| Debin [55] | 238 | ④ | No |

**Table 4.3:** Comparison with datasets available in literature. Evaluated tasks: ① Binary Similarity, ② Function Search, ③ Compiler Provenance, ④ Function Naming, ⑤ Signature Recovery

In the following, we provide an overview of the analyzed datasets.

**Binkit** [69] proposes a dataset for the binary similarity task. The dataset is obtained by compiling 51 GNU packages using 9 different compilers and 5 optimization levels. The dataset is divided in subsets, one for each different compilation option.

**jTrans** [124] introduces a dataset based on C/C++ projects from Arch Linux and Arch User packages. These are compiled with GCC and G++, obtaining 48’130 binary files. Labels are extracted from unstripped binaries, and duplicates are removed to prevent potential biases.

**In Nomine Function** [9] uses AMD64 packages retrieved from Ubuntu 19.04 repository. Duplicates functions were removed as well as functions for which symbolic names cannot be retrieved. The dataset is built for the function naming problem.

$\alpha$**Diff** [81] uses a dataset composed by two different collections of source codes. For each of the projects included in the dataset, different versions have been retrieved. The first collection is composed by 31 projects, compiled using GCC v5.4. The second collection is composed by the binaries of 895 packages from Debian repository.

**SAFE** [87] proposes two datasets for the binary similarity task (one for AMD64 and another for ARM) and one dataset for the function search task. The first training set is obtained by 9 projects for AMD64, compiled with 3 compilers and 4 optimization levels. The second dataset is composed by 2 versions of Openssl, compiled for ARM with GCC v5.4 and 4 optimization levels. For the task of function search, the dataset is composed by packages of AMD64, compiled with 10 compilers and 4 optimization levels.

**Graph Embedding NNs** [86] proposes two datasets for compiler provenance. The first is composed by open-source projects, compiled with 3 different compilers and 4 optimization levels, while the second is constituted by several projects, compiled with 11 compilers.

**TREX** [97] introduces a dataset composed by 13 open-source projects. These are compiled for different architectures, namely x86, x64, ARM (32-bit), and MIPS (32-bit). The packages are compiled using GCC v7.5, with 4 optimization levels (-O0, -O1, O2, O3, O4). For x64 architecture, 5 types of obfuscations are applied on all projects.

**BinFinder** [100] solves the function search on binaries compiled from 7 open-source packages. The approach generates 5 distinct datasets, composed by non stripped binaries. The first dataset is composed by 116’508 functions, compiled for x86 instruction set with GCC, Clang, and O-LLVM, using 4 optimization levels and 3 obfuscation options. The second dataset includes 157’673 functions, compiled with 4 optimization levels for x86 and ARM architectures. The third dataset includes 284’491 functions, and extends the first dataset on ARM and x86. The fourth dataset includes 60’395 functions from 3 C/C++ packages. These are obfuscated with 5 methods, and compiled with GCC for x86 architecture. Finally, the fifth dataset contains 164’700 functions, compiled for x86 architecture by 49 packages with Clang, and O-LLVM. The binaries are generated using 4 optimization levels and 3 obfuscation techniques. The datasets are not distributed.

**Toolchain Provenance** [105] solves the task of compiler provenance on a dataset composed by 8 open-source packages, compiled with 9 versions of 3 compilers, using 2 optimization options. The dataset is not released.

**Asm2Vec** [35] proposes two different datasets for the function search task. The first one is composed by binaries of 10 libraries, generated with GCC v5.4 and 4 optimization levels. The second dataset is composed by a subset of the previous one, composed by 4 libraries. It is built using Clang and 4 obfuscation options of Obfuscator-LLVM. The dataset is not distributed.

**Gemini** [129] solves the binary similarity task. It uses a training dataset, obtained by compiling 2 versions of OpenSSL, using GCC v5.4 in 4 optimization levels. The dataset is not released.

**Eklavya** [22] solves the signature recovery task on a dataset composed by Linux packages, compiled with 2 compilers and debugging symbols.

**NERO** [28] uses a dataset for the function naming task. The dataset is built using packages from GNU repository. To build the dataset, packages containing mixed programming languages were filtered out, and duplicate packages were removed.

**Debin** [55] uses a dataset composed by Linux packages, that is not being released. The packages compiled in multiple optimization levels and different compilers.

## 4.4 Baselines

In the previous section, we have compared our benchmark against existing research. In this section, we want to assess the generalizability of BinBench benchmark. For this reason, we evaluate various architectures on the proposed tasks. Furthermore, whenever possible we use existing pretrained models. The resulting predictions are evaluated with the task-specific metrics and used as baselines of our benchmark.

### 4.4.1 Binary Similarity Baseline

For binary similarity and function search, we use the pretrained model of SAFE [9], implemented with Tensorflow and Python. SAFE computes an embedding in two phases. First, it embeds each assembly instruction using word2vec model [90]. Therefore, it computes the final embedding using a *Self-Attentive Neural Network*. As mentioned earlier, this network is a bi-directional RNN that produces a summary vector for each input instruction. Then, SAFE computes the function embedding as a weighted sum of all summary vectors.

To solve the binary similarity task, we have to evaluate datapoints composed by pairs of functions. Therefore, we first embed each binary function using SAFE. Then, we compute the cosine similarity between embedding pairs. We consider a predefined threshold $T = 0.6$ to convert each cosine similarity into a label. More precisely, a couple is marked as similar (i.e., label = +1), when the cosine similarity is greater than $T$, and dissimilar (i.e., label = -1) otherwise. The result for binary similarity task is reported in table 4.4.

| Area Under the Curve (AUC) |
|---|
| 0.91 |

**Table 4.4:** Result for the binary similarity task

### 4.4.2 Function Search Baseline

To solve function search task, we use the SAFE pretrained model described previously. More precisely, we first embed every binary function in the query list and database. Therefore, we compute binary similarity measure between each embedded query, and each embedded database function. Finally, we analyze the resulting similarity scores. For each query $q$, we retrieve the $K$ most similar database functions (having greater value of the computed measure). The resulting functions are ordered with respect to the grade of

[9] https://github.com/gadiluna/SAFE

similarity (i.e., the first function is the most similar to $q$, the second is the second most similar to $q$, and so on). The described ordering is respected for each function in the query list. The results for the function search task are reported in Table 4.5.

| Precision | Recall | F1 Score | nDCG Score |
|---|---|---|---|
| 0.26 | 0.26 | 0.26 | 0.36 |

**Table 4.5:** Results for the function search task

### 4.4.3 Compiler Provenance Baseline

For compiler provenance, we use a pretrained model [10] of the Graph Embedding Neural Network described in [86]. As anticipated in the background section, this model embeds a CFG graph through two components. The first one is the *Vertex Feature Extraction*, that maps each vertex of the CFG into a feature vector. The second component, the *Structure2Vec network*, uses deep neural networks to produce the final graph embedding. This component creates a vector for every graph vertex. In the training phase, those vectors are dynamically updated using an approach based on rounds. Vector updates take into account graph topology and previously extracted features. Therefore, the graph embedding is computed as the aggregation of updated vectors. The model is implemented using Tensorflow and Python.

To solve compiler provenance task, we first train the model on its own dataset, and then use it to infer the requested datapoints. In detail, the dataset used for the training is the *restricted compiler dataset* , including multiple compiled open-source projects. These packages are compiled for AMD64 architecture using 3 compilers, namely GCC v3.4, GCC v5.0, and Clang v3.9. In addition, 4 optimization levels are used. Resulting binaries are disassembled using *radare2* [11]. The results achieved by the model on out task are showed in Table 4.6.

| Accuracy | Precision | Recall | F1 Score |
|---|---|---|---|
| 0.81 | 0.78 | 0.88 | 0.82 |

**Table 4.6:** Results for the compiler provenance task

### 4.4.4 Function Naming Baseline

To solve function naming task, we use the pre-trained Transformer proposed by In Nomine Function [12]. This architecture represents binary functions as list of normalized instruc-

[10] https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity

[11] https://rada.re/

[12] https://github.com/gadiluna/in_nomine_function

tions. Furthermore, each function name is transformed into a list of tokens.

To solve the function naming task, two architectures are used, namely the Seq2Seq and Transformer. These models are trained over a big dataset for a maximum of 30 epochs with early stopping mechanism, with Adam optimizer and batch size of 512. The model has been implemented using OpenNMT-py.

In our experiment, we evaluate our benchmark with the proposed pre-trained Transformer. However, the performance is lower with respect to the other tasks. This could be caused by the fact that our dataset includes tokens on which the model has not been trained. Baseline results for function naming are reported in Table 4.7.

| Precision | Recall | F1 Score | BLEU |
|---|---|---|---|
| 0.07 | 0.04 | 0.05 | 0.03 |

**Table 4.7:** Results for the function naming task

### 4.4.5 Signature Recovery Baseline

To solve signature recovery task, we do not have any pre-trained model to use as baseline. For this reason, we have trained a Transformer from scratch using the OpenNMT-TF toolkit [13].

We use default configuration of the Transformer architecture, and train it over a portion of our labaled (training) dataset. The training set were composed by 115'000 randomly picked binary functions, where 100'000 were used for training and the remaining 15'000 were used for validation.

For training the transformer, we used batch size equal to 200 and Adam Optimizer. We validated our model every 100 steps, stopping on the highest result for the validation split. The highest performance is achieved by the model trained for 3'850 steps. The results of the transformer are summarized in Table 4.8. It should be noted that for this task the metrics are calculated using the micro averaging method.

| Accuracy | Precision | Recall |
|---|---|---|
| 0.53 | 0.53 | 0.53 |

**Table 4.8:** Results for the signature recovery task

### 4.4.6 Discussion

We have evaluated different architectures over the proposed tasks: binary similarity, function search, compiler provenance, signature recovery and function naming. The selected

[13] https://github.com/OpenNMT/OpenNMT-tf

tasks address significant real-world applications, including malware detection and reverse engineering.

In order to test models on these tasks, we have provided two elements. The first is the set of datapoints to be predicted, and the second is the evaluation method. We evaluated each task with known models, and use their results as baseline scores for the benchmark.

Depending on the task evaluated, different performance is achieved. In particular, for binary similarity we obtained high AUC using the pre-trained SAFE model. We used the same models for the function search task. However, in this case we achieved lower results than binary similarity. This is probably because the function search task is more complex than binary similarity.

For compiler provenance task, we have first trained the graph embedding neural network on its own dataset. Then, we use the model to infer compilers of our datapoints. As showed in the previous section, the model achieves high performance when predicting our datapoints.

For function naming, we used a transformer model that was pre-trained on the same task. However, the model's performance on our datapoints was low. A potential explanation is that our dataset could include tokens not used during the model training. To investigate these results, we aim to train various models from scratch on our training dataset and then evaluate them on our task. We believe that a specific training phase could improve the performance achieved for the task of function naming.

Finally, for the signature recovery task we opted for a transformer architecture. The model was trained from scratch on our dataset, and then used to predict the test datapoints. The model achieved good results that could be potentially improved using different configuration in the training phase.

We want to point out that BinBench is the only benchmark defined for evaluating models over all the considered tasks. Furthermore, to facilitate the training process, the proposed benchmark provides both the binaries and JSON representations of the functions to be evaluated.

# Chapter 5

# Code Smell Detection

The Internet of Things (IoT) is constantly expanding in various fields. As a consequence, the number of digital devices being connected to the internet is growing. On the other hand, with the increasing number of network connections, the risk of cyber-attacks is also rising. In addition, such attacks are becoming more and more sophisticated, and focused on exploiting specific vulnerabilities on targeted devices. A 0-day vulnerability affecting a widely used library can propagate into numerous programs, making the code vulnerable to potential attacks. As a consequence, exploiting one of these vulnerabilities can compromise a vast range of devices, regardless of their underlying architecture. Code analysis is therefore essential in this scenario. The analysis involves the examination of program instructions, functions, and flow of execution. However, bugs are not the only source of vulnerabilities. The code can be also affected by more subtle issues known as *code smells*. These are a combination of wrong design choices, negatively impacting the quality of the source code. An example of code smells is the *Data Class*, a term used for denoting classes focused only on storing attributes without any other meaningful functionalities. Another example of code smells is the *Complex Conditional*, consisting in having an excessively complex and hard-to-read conditional statement in a function.

When code smells are exploited, attackers can, for example, inject specific inputs in the code. This could result in unexpected code executions and subsequent anomalous behavior of the IoT devices on which the code is installed. The early detection of code smells is therefore of paramount importance for securing IoT systems and, in general, for software development. For this reason, different detection techniques have been developed.

```
public class Employee {
        private int userID;
        private String jobTitle;
        private String workLocation;

        public int getUserID() {
                return userID;
        }
        public void setUserID(int userID) {
                this.userID = userID;
        }
        public String getJobTitle() {
                return jobTitle;
        }
        public void setJobTitle(String jobTitle) {
                this.jobTitle = jobTitle;
        }
        public String getWorkLocation() {
                return workLocation;
        }
        public void setWorkLocation(String workLocation) {
                this.workLocation = workLocation;
        }

}
```

**Listing 5.1:** Example of Data Class

Traditional methodologies for code smell detection relies on metric-based thresholds [85, 108], or the verification of predefined heuristics on code features [91, 114]. However, these techniques have limitations. Definition and implementation of detection rules and metrics are complicated and error-prone. For this reason, the application of traditional code smell detection techniques requires many resources, and the availability of experienced personnel. In addition, these detection methodologies are dependent on subjective judgments of the experts developing them. For example, a person can consider a method as "too long" when it is composed by more than 15 lines of code, while others can set the threshold to 20 or 30. This lack of objectivity introduces a further level of complexity in code smell detection. To address these issues, code smell research is exploring alternative methodologies for the detection.

Recent trends are focused on searching valid machine learning techniques for automating the identification of smells in code. Unlike traditional methods, machine learning does not require manual definition of heuristics or metrics. A trained model can automatically identify potential smells in code snippets without human intervention. This offers significant advantages in terms of resource efficiency and objectivity of the detection. The application of these techniques is supported by the naturalness hypothesis [3], suggesting that source code can be considered as a form of human communication.

Software corpora, therefore, have same statistical properties as natural language. This similarity allows to apply natural language processing on the study of code characteristics through the application of machine learning techniques. In particular, the research on applying DNN architectures for performing automated detection of different code smells. Recent researches on this topic show that the approach is very promising [52, 82].

Given the importance of code smells detection for code analysis and refactoring, the thesis's second contribution analyzes the application of various LLMs for the automated detection of different code smells within methods. The contribution also examines how different pre-training approaches influence the detecting performance. This research aims to define a dependable and automated detection technique, that could be dynamically fine-tuned on other types of code smells.

The chapter is split in four sections. The section 5.1 provides an overview of code smell types and current machine learning detection techniques. The section 5.2 outlines the architectures of LLMs selected for our experiments on automated code smell detection. The section 5.3 describes the construction of a comprehensive dataset of different code smells, used to fine-tune and test the selected LLMs on detecting mothod-level code smells. The section 5.4 explains the proposed automated approach, investigating code smell detection using large language models.

## 5.1 Background on Code Smell Detection

Software development is a multi-stage process that involves frequent revisions of source code over time, often performed by different developers. These constant updates could not only introduce inconsistencies but also code smells, representing degradation of code quality in terms of design and implementation [10, 115]. To solve these issues, developers need to refactor their code. Refactoring consists in restructuring the source code to remove the detected issues, without altering functionalities of the software. The complexity of this process requires a significant investment in terms of time and resources.

As showed also in [43], code smell can manifest at different levels within the software. In general, code smells may affect methods, entire classes or even the whole application. In the following, we present a summary of the most significant types of code smells identified in literature. We also provide an overview of code smell detection based on machine learning techniques. In addition, we study different types of large language models, that will be used in our study to automatically detect a variety of code smells on methods.

### 5.1.1 Code Smells

Principal researches on code smell detection are focused on object-oriented programming [42, 113]. Following the same approach, we analyze smells that specifically affect object-oriented code. We categorize code smells into two main groups: *class-level smells* and *method-level smells*.

#### Class-Level Smells

Class-level smells are affecting the design and implementation of entire classes. In the following we provide definitions of most common code smells on classes.

- **Data Class**. This code smell occurs when a class is focused on storing data in attributes, without offering complex functionalities to justify its existence as a class. Data are often exposed using accessor methods.

- **God Class**. It consists in a large and complex class, implementing main functionalities of a software. It also manipulates data from other classes, as for example data classes.

- **Multifaceted Abstraction**. It refers to an abstraction having multiple responsibilities assigned to it. This code smell violates the "single responsibility principle", establishing that an abstraction should have only one responsibility assigned.

- **Refused Bequest**. The smell represents a potential issue regarding how inheritance is being used. It happens when a subclass partially uses the inheritance from its parents.

- **Shotgun Surgery**. This smell occurs when a developer is making change in a class, requiring the modification of multiple parts in the rest of the codebase. These changes can lead to duplications, making the code difficult to be maintained.

#### Method-Level Smells

Method-level smells consists in poor choices regarding design and implementation of functions. Principal smells on methods are the following.

- **Complex Conditional**. This code smell occurs when conditional logic is too complex and difficult to understand. Complex Conditionals increase the likelihood of introducing mistakes in code logic. They also reduce the overall readability of the code.

- **Switch Statements**. Similarly to Complex Conditional, this smell occurs when a method contains statements in a switch that are overly complicated.

- **Complex Method**. It consists in methods having complex logics that are difficult to analyze. As for previous code smells, Complex Method increases the chance of creating errors in code development.
- **Feature Envy**. It refers to a method relying more on functionalities of another class than on the features of its own class. This behaviour can be a sign that the method might be in the wrong place and could be moved into the other class.
- **Long Method**. This code smell happens when a method is too large. Having too many lines of code is error prone, and reduces the understandability of the method.
- **Long Parameter List**. It occurs when the method's parameter list contains too many parameters. This smell increases the complexity of the method, and its readability.

### 5.1.2 Machine Learning Techniques for Code Smell Detection

A key challenge of code smell identification is obtaining a sufficient number of examples for training and testing detection models. Identifying code smells can be subjective due to their varied interpretations, and generally require manual intervention. For this reason, the creation of a comprehensive collection of different code smell samples is not an easy task. However, these datasets are required to train and evaluate detection models.

For this reason, **Arcelli Fontana et al. [42]** create and distribute a dataset containing different code smells. The collection was originally composed by four smells, namely Data Class, Large Class, Feature Envy and Long Method. After the publication of [42], the dataset has been expanded with two further code smells, Long Parameter List and Switch Statements.

To extract code smell examples, researchers use a collection of software, known as Qualitas Corpus version 20120401r [118]. From this dataset, including code in Java language, researchers extract 74 heterogeneous software systems, containing roughly 320 packages with 52'000 classes and 404'000 methods. To find proper samples within these systems, researchers apply various deterministic rules (i.e., "Advisors") through external tools. The advisors are used to automatically approximate labels, indicating whether a code snippet is affected by a specific smell. For each code smell, researchers identify multiple advisors. In addition, researchers apply a stratified sampling on code snippets to avoid potential biases coming from the advisors.

Each code candidate is associated to two values, the name of its original project and the number of Advisors considering it as a positive instance (i.e., smelling code). Candidate codes are then grouped with respect to these two values. The process cycles over the groups and randomly selects one item from each group at a time. The element selected is evaluated by human experts, manually labeled, removed from the group and finally added to the training set. The labels assigned indicates whether a code is affected by a specific code smell.

Random sampling is repeated until the number of required elements is reached. The ratio of positive to negative instances is adjusted to reflect the real-world scenario, where specific code smells are less frequent than "correct" code. Using this process, researchers created one dataset for each code smell, each containing 420 examples.

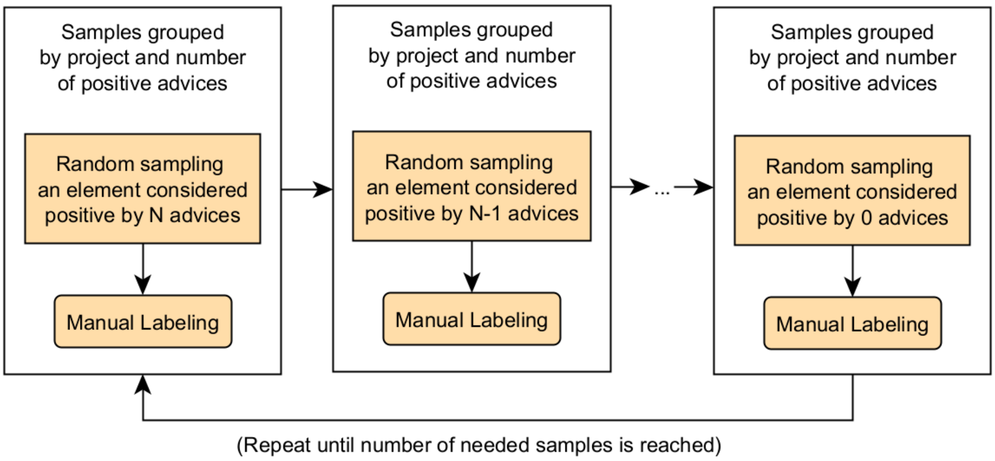
Samples grouped by project and number of positive advices
Random sampling an element considered positive by N advices
Manual Labeling
Samples grouped by project and number of positive advices
Random sampling an element considered positive by N-1 advices
Manual Labeling
...
Samples grouped by project and number of positive advices
Random sampling an element considered positive by 0 advices
Manual Labeling
(Repeat until number of needed samples is reached)

**Figure 5.1:** Process of random sampling used in [42]

In parallel to candidate selection, researchers also compute a set of predefined and custom metrics, calculated at different levels of granularity in the software. These metrics refer to various features of the code, such as complexity (e.g., CYCLO, WMC, ect.), size (e.g., LOC, NOM, etc.) and inheritance (e.g., DIT, NOI, etc.). These metrics are used as independent variables for machine learning techniques.

Researchers evaluated a variety of machine learning algorithms on code smells collected, such as J48, Random Forest and Naive Bayes. These algorithms have been validated on each code smell using 10-fold cross validation. This approach randomly split data in 10 folds of same size. One fold is used as test set, while the rest as training set. The approach is iterated, and each time a different fold is used as test set. The research show that best performance for code smell detection are achieved by J48, reaching 97% of accuracy.

In [122], **Walter et al.** use a similar approach to create and publish a dataset, extracting 92 Java-based systems from Qualitas corpus version 20130901. The dataset contains 14 different code smells, including Refused Bequest, Feature Envy, Long Method, Shotgun Surgery, Long Parameter List, God Class, Data Class, and Data Class.

The identification of smelly codes is performed using automatic tools. Researchers defined three "working datasets", including instances that were detected by at least 25%, 50% and 75% of the automatic tools respectively. These datasets are inclusive: code examples in the 25% are also included in the 50% and the 75% datasets. Each datapoint is represented by an entire class affected by at least one code smell. In addition, researchers associate a binary vector to each code snippet, indicating whether one or more automatic tools have detected specific types of code smells on it. The table [5.1] shows an example of a binary vector associated to a smelly class in the dataset. For simplicity, only 4 out of 14 available columns have been reported in the table.

| Feature Envy | Long Method | Shotgun Surgery | God Class | ... |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | ... |

**Table 5.1:** Example of binary vector associated to a sample of the dataset defined in [122]

The dataset has been used for studying relationships between different types of code smells using pair-wise correlation analysis, Principal Component Analysis and associative rules.

**Madeyski et al.** [84] define and distribute the MLCQ dataset for training machine learning models on code smell detection. MLCQ contains more than 15'000 Java code snippets, extracted from 792 GitHub projects.

The dataset contains four types of code smells, namely Feature Envy, Long Method, God Class, and Data Class. These samples were manually reviewed by expert developers. Each datapoint of the dataset represents a review of a code snippet. For each review, researchers collect various information, including identifier of the reviewer, timestamp of the review, type of code smell, smell label, datapoint's identifier, and the link to relevant source code. The smell labels indicate the severity of a specific smell in the code analyzed. The labels are defined on a scale from 0 (i.e., the code is not smelly) to 4 (i.e., the code is critically impacted by a particular code smell).

In [96], **Pecorelli et al.** propose a comparison of heuristic methods and machine learning techniques for metric-based code smell detection. The study investigates five different types of code smells, namely God Class, Spaghetti Code, Class Data Should Be Private, Complex Class, and Long Method. The experiments are performed on a dataset containing 8534 manually validated code snippets.

Researchers define a set of detection rules, one for each code smell, and apply them using an automatic tool. This heuristic-based approach is used as baseline for machine learning classifiers. For instance, the detection rule of Long Method is the following.

```
LOC Method > 100 AND NP > 1
```

Where LOC Method represents the number of lines of code in a method, while NP is the number of parameters. Five machine learning algorithm were selected for the experiments:

J48, Random Forest, Naive Bayes, Support Vector Machines and JRip. The models are validated using 10-Fold Cross Validation.

### Deep Neural Networks for Code Smell Detection

**Hadj-Kacem and Bouassida** [52] propose an hybrid approach based on deep neural networks. The code smells analyzed in the study are God Class, Data Class, Feature Envy and Long Method. The process consists in two steps. First, it employs an unsupervised learning using auto-encoder architecture. Then, the approach uses a supervised learning for classification with neural networks.

The auto-encoder is a neural network composed by two parts. The encoder that applies transformations on the input, and the decoder reconstructs the original input from the encoder representation, creating an equivalent but lossy output. This process allows to reduce the dimensionality of the given input. For this reason, auto-encoders are often used for extracting features to inject in another network. Features extracted by the auto-encoder are used as input for an artificial neural network trained with supervised approach. The network is trained to perform binary classification. It learns to predict labels that indicate whether a piece of code is affected by a specific code smell.

The approach is evaluated on the dataset proposed by Arcelli Fontana et al. [42]. For the experiments, researchers define an auto-encoder composed by input layer, three hidden layers, and output layer. The architecture is trained using mean squared error as loss function, and hyperbolic tangent as activation function. The proposed approach is evaluated using 10-fold cross validation. The model show significant results in detecting different code smells. Highest precision is reached for God Class (99.28%), while the lowest is obtained with Feature Envy and Long Method (96.78%).

**Liu et al.** [82] propose another approach for automatic detection of code smells using neural networks. The study investigates four code smells: God Class , Misplaced Class (classes placed in incorrect packages), Feature Envy, and Long Method.

The training dataset is generated by refactoring a large collection of open-source projects in Java language. Researchers assume that this corpora is well-designed and therefore not affected by code smells. For each code smell, researchers select all available code snippets in the collection on which is possible to apply the refactoring. Then, the approach randomly picks one of these codes and apply the refactoring. The resulting code is impacted by the specific smell generated with the refactoring. Therefore, the code is added in the positive samples. The process cycles until the number of required smelly instances of code is reached. For negative samples, the method randomly selects code snippets from the original software corpus without duplication. The collection of negative samples is based on the assumption that code in the original corpora is well-written

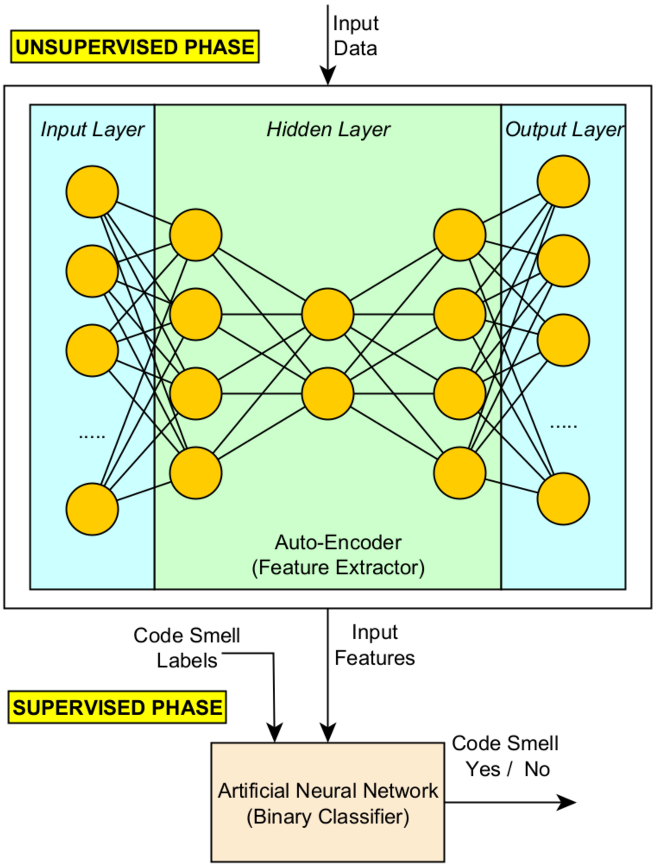

**Figure 5.2:** Hybrid architecture for code smell detection proposed in [52]

and free from code smells. However, the hypothesis could not hold for every class and method in the software corpus. As a consequence, some negative example could still be impacted by code smells, creating noise within the experiments. At the end of the process, researchers obtain four datasets, one for each code smell. From these datasets, researchers extract N subsets each; these are composed by samples that have been randomly selected.

For each code smell, researchers use the N code smell subsets to train in parallel N neural network classifiers. During testing, each code example is fed as input to the N classifiers. Each network produces a binary classification of the code. Then, the final choice is determined through a voting process. For each type of code smell, researchers propose a different architecture for the N classifiers. For instance, the classifiers trained on Feature Envy take two inputs, a text sequence and a pair of numerical values.

The textual input is composed by the name of the method considered, the name of its actual class, and the name of the feasible destination class. The information is transformed by an embedding layer. Then, the resulting feature vector is fed to a Convolutional Neural Network (CNN), which is an architecture used for feature extractions. The output of the CNN is fed into a flatten layer, transforming inputs in one-dimensional vectors.

The numerical input is composed by the distance between the method considered and its actual class, and the distance between the method and a potential destination class. The distances are computed as follows.

$$distance(method, actual - class) = 1 - \frac{S_{method} \cap S'_{class}}{S_{method} \cup S'_{class}}, \quad S'_{class} = S_{class} - \{m\} \tag{5.1}$$

$$distance(method, potential-destination-class) = 1-\frac{S_{method} \cap S_{class}}{S_{method} \cup S_{class}}, \quad S_{class} = \bigcup_{e_i \in C} e_i \tag{5.2}$$

Where $S_{class}$ and $S_{method}$ are the set of entities in the class, and the set of entities accessed by the method respectively. These two distances are fed into a CNN. The output is then ingested by a flatten layer.

The transformed numerical and text inputs are then taken as input by a merge layer, followed by a dense layer. Finally, the output layer produce a binary prediction, indicating whether the method is affected by feature envy.

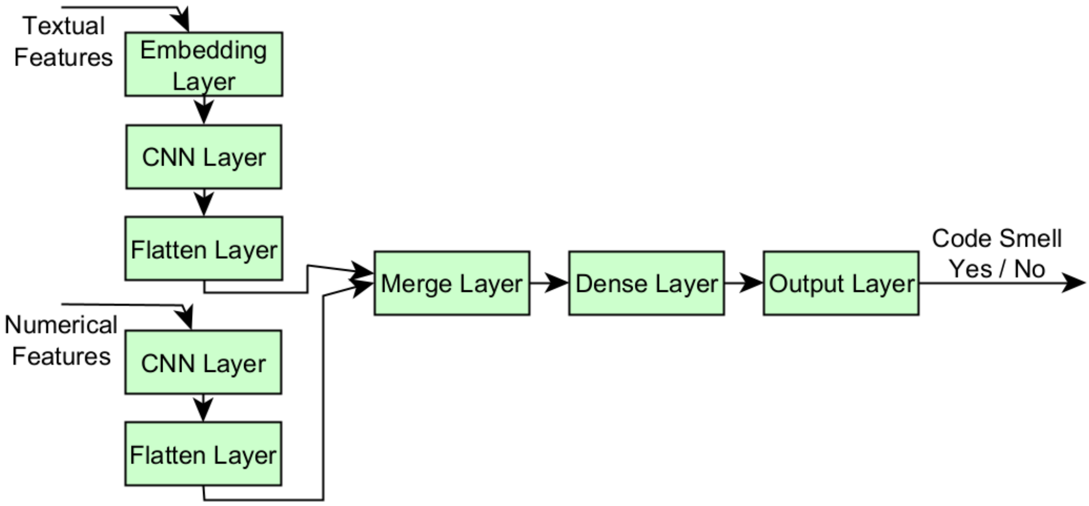

**Figure 5.3:** Architecture of the classifier for detecting feature envy defined in [82]

For detecting Long Method, the architecture proposed includes an input layer, five dense layers, and an output layer. For each method considered, the classifier takes in input a variety of metrics on the code. In output, the model returns a binary classification of the method considered. The prediction represents whether the input is affected by Long Method code smell.

For Large Class, the architecture of the N classifiers takes as input a set of code metrics, and a text sequence, composed by method name and field names. Code metrics are ingested by a dense layer, while the textual features are fed into an embedding layer, and then forwarded to a LSTM layer. Outputs of the dense layer and the LSTM layer are passed to a merge layer, and a second dense layer. Then, the output layer retuns a binary prediction, indicating the presence or absence of Large Class code smell in the analyzed method.

For Misplaced Class, the architecture proposed for the code smell detection takes two different inputs. A set of code metrics and a textual sequence, composed by class name, actual package, and potential destination package. The first input is fed into a CNN layer, and forwarded to a flatten layer to produce a one-dimensional vector in output. The latter input is ingested by an embedding layer, a CNN layer and finally by a flatten layer to produce one-dimensional vector. The outputs of the two flatten layers are then fed into a marge layer, a dense layer and finally an output layer. The prediction produced shows whether the class is affected by Misplaced Class.

Researchers evaluate the approach proposed on a set of applications. It achieved the greatest accuracy (81.60%) for detecting Misplaced Classes. The lowest accuracy (12.95%) is instead reached while predicting Large Classes.

**Sharma et al.** [113] propose another approach for code smell detection with deep learning models. The method is performed with two different learning approaches, namely direct-learning and transfer-learning. The first approach is used to train a model from scratch a specific task. The latter approach instead involves training a deep neural network for one task, and then leverages the acquired knowledge to better perform on a related task.

The work analyzes four different code smells on object-oriented languages: Complex Method, Complex Conditional, Feature Envy, and Multifaceted Abstraction. The code smells are divided in three categories, which spans across methods and classes. The first is the *implementation code smell*, occurring when issues are introduced in limited parts of the code, such as methods. Examples of implementation smells are Complex Method, and Complex Conditional. The second is the *design code smell*, an issue involving one or more classes. Examples of design smells are Feature Envy and Multifaceted Abstraction. Finally, the *architecture code smell* refers to a design issue affecting different elements of a system. An example of architecture smell is Scattered Functionality, where functionalities related to a specific concept are dispersed across multiple components of the system.

To test their approach, researchers construct and distribute a large dataset of code smell samples. The code collection is composed by 922 C# and 1721 Java repositories extracted from GitHub. To ensure the accuracy of the labeling process, the work utilizes automated tools that have been initially validated manually. During the experiments, samples are divided in training and test sets using a ratio of 70%-30%. Training and test sets are balanced to avoid potential biases. The train set includes a limited number of positive and negative instances, with ratio of 50%-50%. For the test set, researchers apply limitations on the number of positive and negative examples used.

The approach is evaluated using three different types of neural networks: CNN, RNN and auto-encoder. Each model is trained in direct learning and transfer learning. In the latter case, the model is first trained on C# or Java code snippets, then evaluated on detecting code smells in samples written in the other language. The input code snippets are converted in numerical vectors using a tokenizer and then fed into the DNNs.

The first architecture proposed is a CNN, composed by multiple layers. The input tokens are fed into a convolution layer, then forwarded to batch normalization and max pooling layers. This set of layers is used for feature extraction. The output of the max pooling layer is then fed into a dropout layer to reduce over-fitting. Then, the information is forwarder to a flatten layer and two dense layers. The last layer produces a binary prediction for the code taken as input. Researchers created three different configurations of the model by repeating the feature extraction component 1, 2 or 3 times. To avoid over-fitting, early stopping technique is applied as further regularization.

The second model proposed is a RNN architecture, consisting in a initial embedding layer, followed by LSTM, dropout and dense layers. The final layer produces a binary classification of the code in input. The LSTM layer is repeated 1, 2, or 3 times to create different configurations.

The third architecture proposed in this work is an auto-encoder, for which different configurations are defined. A basic configuration is composed by four dense layers. A more complex setup uses three LSTM layers and a final dense layer. Another complex configuration uses multiple convolutional layers, along with max-pooling and upsampling layers, followed by a final dense layer. In addition, configuration is modified by repeating 1 or 2 times the encoder and decoder parts as showed in figure [5.4].

Experiments show performance variations depending on the code smell type. The highest precision is achieved by auto-encoder (53%) while detecting complex methods. Researchers also compare direct-learning and transfer-learning approaches. Models trained with direct learning generally achieve higher or equivalent performance compared with those trained with transfer learning.

**Kovacevic et al.** [71] evaluate the quality of code embeddings learned by three pre-trained DNN models, namely code2vec, code2seq and CuBERT [64], for detecting God Class and Long Method. Each vector representation is then used to train different machine learning classifiers with supervised learning approach. In addition, researchers compare the approach with heuristic-based methods for code smell detection, and machine learning classifiers trained on code metrics. Experiments are performed on Java code snippets from MLCQ [84], a dataset described in the section 5.1.2.

For the first experiment, researchers use code2vec [5], a deep neural model producing

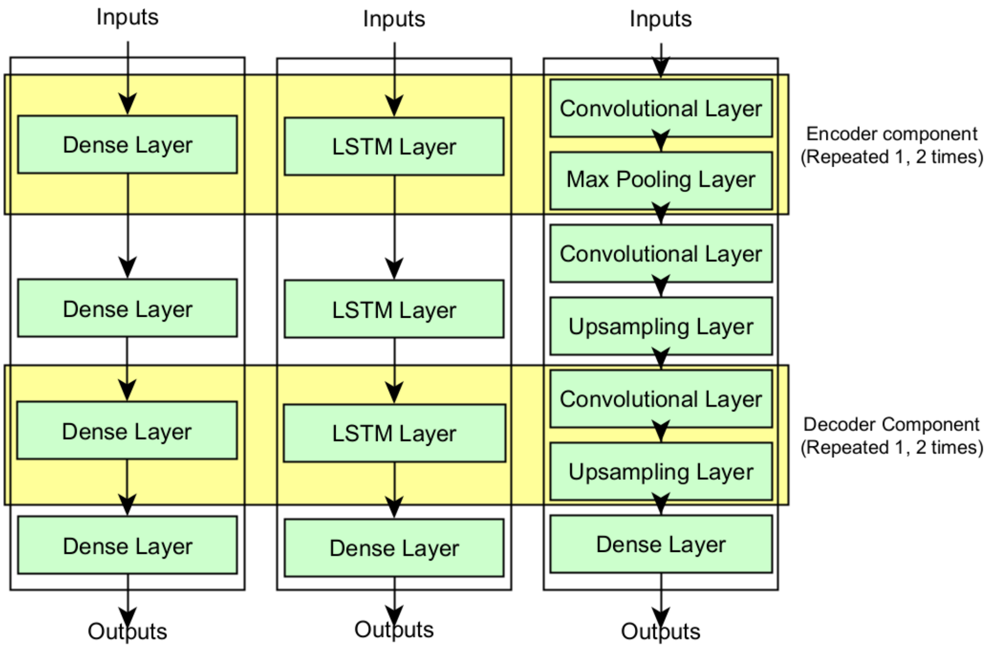

**Figure 5.4:** Autoencoder configurations proposed in [113]

a fixed-length vector embedding for the code taken as input. The model learns the vector representation of paths in the abstract syntax tree associated to the code snippet. Researchers therefore extract AST paths of the methods to be embedded, and feed them into the pre-trained code2vec model. The output embeddings are forwarded to a machine learning classifier, validated through 5-fold-cross validation on the training set.

The second experiments consists in producing a vector embedding with code2seq [4], an encoder-decoder architecture that learns the code representation from AST paths of the input code snippet. The encoder part of code2seq transforms each AST path into a fixed-length sequence of AST nodes. The attention mechanism in the decoder component focuses on relevant paths for creating the output sequence. The resulting embedding is then fed into a machine learning classifier that is validated with 5-fold-cross validation on the train dataset. Code2vec and code2sec are however expected to take as input only method codes. For this reason, researchers generate their class representations by embedding each method separately. Then, class embeddings are computed as the mean of the obtained method embeddings.

The third experiments uses code embeddings provided by CuBERT [64], a LLM deriving from the transformer-based architecture known as BERT [33]. Details of CuBERT architecture are provided in the section 5.2. CuBERT is fed with sequences of code tokens, pre-processed with a specific Java tokenizer. To produce methods' embeddings, researchers fed each line of code into CuBERT, and generate an embedding for each line. The resulting vectors are then summed together to obtain the method representation. The class embedding vector is instead computed as the sum of invidual method embeddings. Then, the approach forwards CuBERT's code embeddings to a machine learning classifier, which is evaluated performing a 5-fold-cross validation.

Researchers compare the approach proposed with different techniques based on heuristics and metrics. For heuristic-based detection, researchers select multiple code metrics on classes and methods, such as the number of methods in a class, and the code lines in a method. Metrics are extracted using open-source automatic tools. The work evaluates the performance of single detection rules and their aggregation.

Researchers also evaluate the performance of classifiers pre-trained on code metrics. The experiment uses a set of code metrics extracted with automatic tools. The experiment also trains various classifiers on a vector representation of input codes. These vectors are composed as the concatenation of code metrics and the related vector of heuristics votes. For each heuristic, the vector of heuristics votes is set to 1 if the code is affected by code smell according to the heuristic considered, and 0 otherwise.

Experiment results show that the machine learning model based on CuBERT's code embeddings achieves highest performance with respect to the other approaches. In detail, heuristic-based method achieves F1-score of 0.49. Machine learning algorithms based on code2vec and code2sec embeddings achieved F1-score of 0.26 and 0.41 respectively. Depending on the configuration used, classifiers based on code metrics reaches F1-score of 0.52 and 0.51. Finally, the machine learning classifier based on CuBERT's code features reaches a maximum F1-score of 0.75, achieved while detecting Long Methods.

In our study, we use a similar approach to detect different method-level code smells. In detail, we use four large language models, namely CuBERT, CodeBERT, GraphCodeBERT, and BERT to learn the representation of code smells on source code. We fine-tune models on each code smell as a binary classification task and then evaluate their performances on test sets.

**Zhang et al.** propose DeleSmell, [133] an approach for detecting code smells based on DNNs. The code smells analyzed are Brain Method and Brain Class, affecting methods and classes concentrating most of the logic of the codebase. The dataset used in the research is automatically generated by refactoring 24 GitHub well-designed projects.

The work extract two differents features: semantic and structural. For semantic features, DeleSmell creates vector features from methods through word2vec model. Furthermore, the approach evaluates cohesion of source code by computing average concept similarity (ACSM) using latent semantic analysis. Higher values of ACSM indicate major cohesion within method or classes and potential presence of Brain Class or Brain Method code smells. For structural features, the approach uses an automatic tool to extract several metrics on methods, classes, packages, and projects.

Input features are fed into two parallel components of the deep architecture proposed.

The first part is composed by a Gated Recurrent Unit (GRU) attention layer, a type of RNN architecture capturing long-term dependencies of sequential data and using additional an attention mechanism to focus on relevant elements of input sequences. The second part is a CNN, used for feature extraction and classification. Outputs of the two parallel parts are fed into a dense layer and forwarded to a final SVM layer, producing a prediction for the input code snippet. The architecture of DeleSmell is illustrated in the figure 5.5.

The work compares DeleSmell model with a variety of machine learning algorithms. The experiments show that the model proposed outperforms other approaches, reaching F1-score of 98.05% and 98.81% in detecting Brain Classes, and Brain Methods respectively.

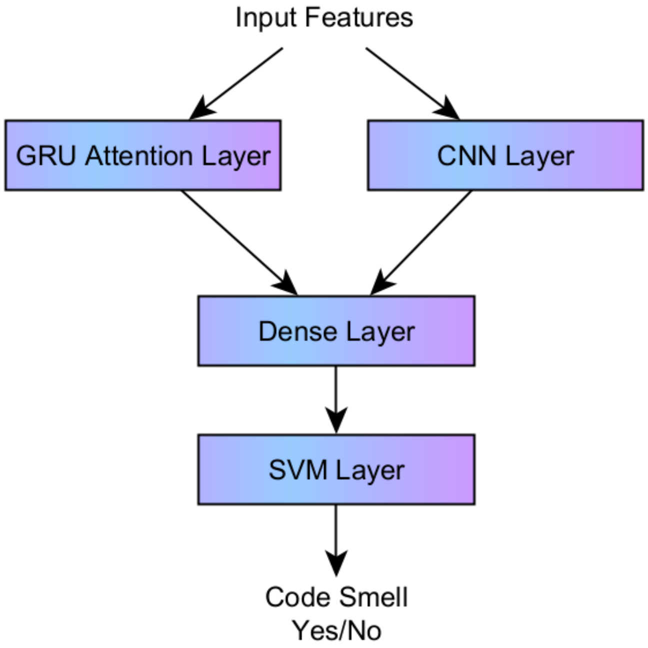

**Figure 5.5:** DeleSmell architecture proposed in [133]

**Ho et al.** [59] propose DeepSmells, an approach for detecting code smells based on CNNs and LSTMs. Four different code smell types are investigated: Complex Method, Complex Conditional, Feature Envy and Multifaceted Abstraction. The dataset used is the one proposed by Sharma et al. [113]. Input information is first pre-processed with tokenization and padding. The resulting input data are fed into the model proposed.

The architecture of DeepSmells consists of two components: feature extraction and classification. The feature extraction part is composed by two convolution blocks, employing ReLU activation function and batch normalization, and a LSTM layer. The feature obtained are then fed into a fully connected network, having one hidden layer and using ReLU activation function. The latter network acts as a binary classifier for each code smell analyzed. Researchers also define two configurations of the DeepSmells, using uni-direction LSTM and bi-directional LSTM.

The experiments performed show that the architecture employing uni-directional LSTM performs slightly better than the other one. For Complex Method detection, DeepSmells' unidirectional LSTM setup achieves a F1-score of 0.75 while the bidirectional LSTM setup achieves F1-score of 0.73. For Complex Conditional, the bidirectional LSTM configuration performs slightly better, reaching F1-score equal to 0.59, while the unidirectional configuration 0.58. For Feature Envy, unidirectional LSTM configuration of DeepSmells reaches a F1-score of 0.29, while bidirectional LSTM setup 0.27. Finally, for Multifaceted Abstraction, the unidirectional LSTM setup of DeepSmells achieves F1-score of 0.27, while bidirectional LSTM version reaches F1-score of 0.26.

## 5.2 Background on Large Language Models

In the following we present four different architectures of large language models, namely BERT, CodeBERT, GraphCodeBERT, and CuBERT. These LLMs are used in our study for detecting various code smells that can impact methods of a source code. While BERT is trained on NLP tasks, the other models are trained on tasks related to source code.

### 5.2.1 BERT

Devlin et al. [33] introduce a powerful large language model named *Bidirectional Encoder Representations from Transformers* (BERT), based on Transformer architecture. Achieving state-of-the-art performance in a variety of NLP tasks, BERT is considered as a key architecture for the NLP community.

BERT is a bidirectional Transformer architecture, based only on encoders components. Two different configurations of the models are provided, varying in the number of layers, the hidden size, and the number of self-attention heads. In particular, the BERT Base setup has 12 layers, hidden size equal to 768 and 12 self-attention heads. BERT Large is instead composed by 24 layers, 12 self-attention heads and having hidden size of 1024.

The input sequences are embedded through WordPiece tokenizer algorithm. The tokenizer splits words into sub-words using a vocabulary of 30'000 tokens. This process is required to better manage out-of-vocabulary words. Furthermore, the *classification token* ([CLS]) is added at the beginning of each tokenized sentence. When the model processes a word sequence, the internal representation of [CLS] is considered as an aggregate representation of the entire sentence in input. This approach is used to summarize word sequences for classification tasks. In addition, BERT can accept as input either a single sentence or a pair of sentences. In the latter case, the two sentences are embedded together as a single sequence. The *separation token* ([SEP]) is used to divide the two sentences.

The input representation of each token in BERT is composed as the sum of three embeddings:

- *Token embedding*, produced with WordPiece algorithm.
- *Segment embedding*, representing the afference of the token to the first or the second sentence of the sequence pair.
- *Position embedding*, indicating the position of a token within the input sequence.

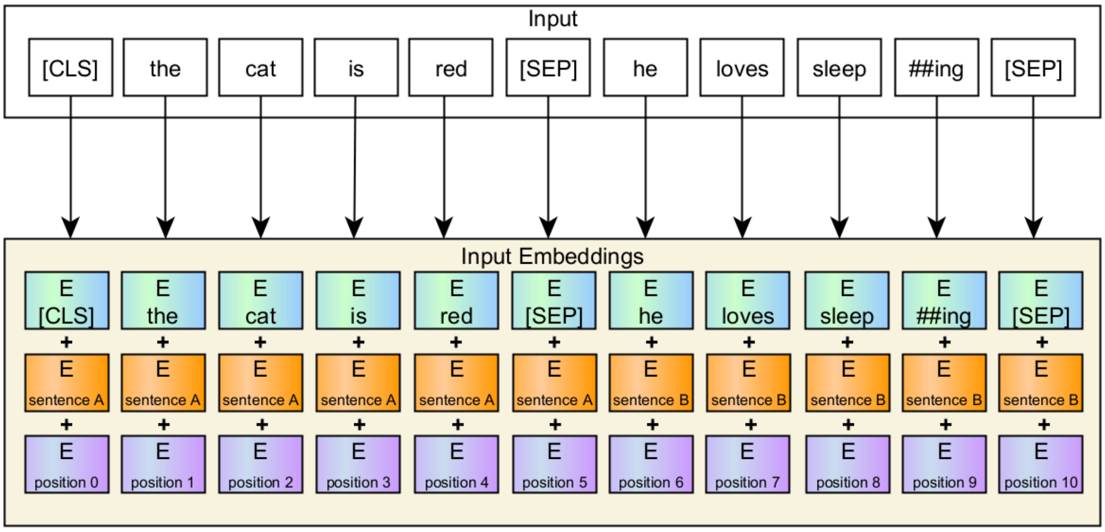

**Figure 5.6:** Input representation of BERT [33], encoded as the sum of token embeddings (green blocks), segment embeddings (orange blocks), and position embeddings (purple blocks).

**Pre-Training and Fine-Tuning** BERT model is first pre-trained on different tasks with an unsupervised approach. Then, the model is fine-tuned on specific tasks using supervised approach. These two phases improves the performance of BERT and facilitates the adaption of the model to downstream tasks.

For the pre-training phase, two tasks are considered. The first task is the *Masked Language Modeling* (MLM), used to learn a bidirectional representation of the input. This approach allows to better understand the context of a word by analyzing information from both directions simultaneously. The task randomly masks part of the input tokens using the [MASK] special token. Then, the model is asked to make a prediction of the masked tokens. In detail, the approach selects 15% of tokens in training dataset to be masked. The 80% of the selected tokens are replaced with [MASK], the 10% of them are substituted with a random token, and the remaining 10% are left unmasked. The second task considered during pre-training is the *Next Sentence Prediction* (NSP), focused on training BERT for capturing relationships between two sentences. The model is presented with pairs of sentences. For each pair, BERT is asked to predict whether the second sentence actually follows the first one. The training set is split equally into positive and negative examples. Positive instances are pairs where the second sentence is actually following the first one in the original corpus. Negative samples, instead, are generated by substituting the second sentence in a pair with a random sentence from the corpus. During training, samples are extracted from two different corpora, namely the BooksCorpus (containing 800M words) and English Wikipedia (including 2500M words).

The second phase of the approach consists in fine-tuning BERT on downstream tasks. Researchers propose 11 different natural language processing tasks to evaluate BERT. This process involves feeding the model with task-specific data and fine-tuning all its parameters accordingly. One of the fine-tuning tasks is the GLUE benchmark, described in section 4.1.2. The process of fine-tuning is performed on all GLUE's tasks in 3 epochs and with batch size of 32. Results show that BERT achieves state-of-the-art performance for GLUE, reaching the highest accuracy (94.91%) with the evaluation of SST-2 dataset.

Researchers provide two pre-trained configuration of the models to be fine-tuned on downstream tasks, namely BERT Base and BERT Large. These configurations are used to test BERT in various tasks, including the source code analysis.

### 5.2.2 CuBERT

Kanade et al. [64] propose *Code Understanding BERT* (CuBERT), a pre-trained model based on BERT architecture. The model focused on learning contextual embeddings for source code. While initially pre-trained on Python as stated in the paper, CuBERT has been expanded also to Java language.

CuBERT is built by training a configuration of BERT Large, composed by 24 layers, 16 self-attention heads, and with hidden size of 1024. The model takes as input logical code lines, intended as minimal line sequences defining legal statements. For instance, a logical code line is a code snippet enclosed in parentheses.

The approach proposes a specific tokenization for code snippets. The input sequences are tokenized with a standard language-specic tokenizer. During tokenization, keywords of the programming language are left unchanged. Special tokens are introduced for denoting code parts that cannot be represented as a string, or having an unclear representation. In addition, identifiers and literals are segmented using predefined rules. A fixed length is also defined for the resulting tokens. This process produces a program vocabulary, that is then encoded to obtain a subword vocabulary. Both vocabularies are used by the model.

Input codes are first encoded with the program vocabulary. Then, the computed tokens are split using the subword vocabulary.

**Pre-Training and Fine-Tuning** CuBERT model is pre-trained on source code extracted from public GitHub repositories of BigQuery's projects. The dataset has been filtered from projects with restricted or unknown licensing. Data are further refined by removing code examples that were highly similar to those included in the fine-tuning set. The similarity is computed with the Jaccard similarity coefficient between set of tokens, defined as follows.

$$Similarity(setA, setB) = \frac{|setA \cap setB|}{|setA \cup setB|} \tag{5.3}$$

Two sets of tokens are considered similar when Similarity(setA,setB)>0.8 or Similarity(setA,setB)>0.7 for multi-sets of tokens. The pre-training is iterated for 1 epoch. The process uses default settings for BERT Large, while varying subword token size (128, 256, 512, 1024) and batch size (8192, 4096, 2048, 1024).

The dataset used for fine-tuning phase is generated from a large collection of GitHub projects. Duplicated code and projects with restricted or unknown license are removed from the dataset. During fine-tuning, the model is evaluated on six different tasks on source code. One task is *Variable-Misuse Classification.* The model is asked to predict if there are variables used in incorrect locations of a given function. Another task is *Wrong Binary Operator*, where the goal is to predict if a binary operator is used correctly within a given function. The task of *Swapped Operand* instead asks to understand whether the operands of a non-commutative binary operator have been swapped. In *Function-Docstring Mismatch*, the model is provided with a pair composed by a function and a docstring. The goal is to identify whether the provided docstring explains the associated function. The task of *Exception Type* requests to find which type of exception is used in a given function. Finally, in *Variable-Misuse Location and Repair* the model is asked to predict two pointers in a function. The first pointer is used to find the position of a variable-misuse, while the second refers to the correct variable to be substituted in the first location.

CuBERT has been fine-tuned on these tasks for 2, 10 and 20 epochs and compared with a transformer trained from scratch in 100 epochs. Results show that CuBERT outperforms performance of the transformer. CuBERT reaches its highest accuracy in the tasks of Function-Docstring Mismatch (98%) and Variable Misuse (95%). In these tasks, the transformer architecture achieves accuracy of 91% and 78% respectively.

### 5.2.3 CodeBERT

Fent et al. propose CodeBERT [41], a pre-trained model deriving from an optimized version of BERT called RoBERTa [83]. CodeBERT is a bimodal model, as it is trained to handle both natural language and code sequences. The model is able to represent a variety of programming languages, including Java and Python.

CodeBERT is built upon the base configuration of RoBERTa model, having 12 layers, 12 self-attention heads, and hidden size of 768. The model produces in output a contextual embedding of programming language, natural language and [CLS] tokens.

**Pre-Training and Fine-Tuning** During pre-training, CodeBERT takes as input a sequence composed as follows.

$$[CLS], w_1, ..., w_n, [SEP], c_1, ..., c_m, [EOS]$$

Where [CLS] and [SEP] are special tokens used in BERT, $w_1, ..., w_n$ represent natural language tokens, $c_1, ..., c_m$ are code tokens, and [EOS] is the *end of sequence* token. The model is pre-trained on a large dataset of GitHub projects, containing bimodal and unimodal data of six different programming languages, namely Java, Python, JavaScript, Ruby, PHP, and Go. Bimodal information refers to paired samples composed by a programming language function and its corresponding documentation. Unimodal examples instead are only composed by programming language functions. The samples are pre-processed by removing short or truncated data.

CodeBERT is pre-trained on two tasks, *Masked Language Modeling* and *Replaced Token Detection* (RTD). MLM is a bimodal task, where the model is asked to predict the value of a masked token. The masking is performed randomly on both natural language and programming language tokens. For RTD, two different configurations are defined: bimodal and unimodal. In this task, the model has to find if a token has been replaced with another token. The replacement is performed randomly in every position of the input. Researchers employ two n-gram models for generating the replacement tokens. One model produces substitutions for natural language tokens, while the other generates replacements for code tokens.

CodeBERT is fine-tuned on various tasks focused on both natural languages and programming languages. In *Natural Language Code Search*, the model has to find in a collection of code the one that is the most semantically relevant for a given natural language sequence. The input for this task is provided in the same way as the pre-training phase; the [CLS] token is used to evaluate the relationship between code and natural language sequences. CodeBERT is also fine-tuned for the task of *Code to Documentation Generation*. In this task, the model takes as input a code and has to generate a relevant documentation. In addition, the model is tested on the zero-shot problem of *Task Formulation and Data Construction*. Given a bimodal pair of data, the model has to retrieve a masked natural language or code token from a set of choices, called distractors.

CodeBERT shows significant performance for all the tasks. The model achieves an overall BLEU score of 17.83 for Code to Documentation task, an overall accuracy of 86% in Task Formulation and Data Construction and an average accuracy of 76% for Natural Language Code Search.

### 5.2.4 GraphCodeBERT

GraphCodeBERT [51] is a pre-trained model based on BERT architecture. The model learns a programming language representation based on the *data flow graph*, representing the semantic structure of the code. The pre-trained configuration of GraphCodeBERT is composed by 12 layers, with 12 self-attention heads and hidden states dimension of 768.

In detail, the data flow shows dependency relationships between variables in a source code. In the graph, edges indicate the origin of variable values, while nodes represent the variables themselves. To generate the data flow graph, the approach first extracts an AST of the source code using an automatic tool. Each variable of the AST is then represented as a node in the data flow graph. For each value assignment for variables in the code, the approach adds a directed edge in the data flow graph.

The input of GraphCodeBERT is a concatenated sequence composed as follows.

$$[CLS], w_1, ..., w_n, [SEP], c_1, ..., c_m, [SEP], v_1, ..., v_q$$

Where [CLS] and [SEP] are special tokens defined in BERT, $w_1, ..., w_n$ are tokens representing natural language comments of the code, $c_1, ..., c_m$ are code tokens, and $v_1, ..., v_q$ are the variables in data flow graph. The model encodes input sequences into input embeddings. For each input token, the input embedding is obtained as the sum of the token and its position embedding. Researchers also define a function for graph-guided masked attention, used to focus only on relevant signals, while removing the others.

**Pre-Training and Fine-Tuning** GraphCodeBERT is pre-trained on three different tasks. The first is *Masked Language Modeling*, where 15% of code and natural language tokens are randomly substituted. The replacement policy used is the one proposed in BERT for the related MLM task. The goal of the task is to retrieve the original value of masked tokens. When the code context is not useful for retrieving a masked code token, GraphCodeBERT can be supported by the context of related natural language comments. This approach facilitates the alignment of code and comment representations. The second pre-training task is *Edge Prediction*, used to train the model on data flow representations. The approach randomly selects 20% of data flow nodes, and then mask edges that links the selected nodes. The task asks to retrieve the masked edges. Finally, the third pre-training task is *Node Alignment*, in which the model learns to align representations of the code and the data flow. The method picks 20% of data flow nodes and then masks edges that link code tokens to these nodes.

GraphCodeBERT is fine-tuned on four tasks. In *Natural Language Code Search*, the model takes as input a comment and has to retrieve the most semantically relevant source code from a corpus. In the task of *Code Clone Detection*, GraphCodeBERT has to evaluate the similarity score between two code snippets. In the task of *Code Translation*, the model is asked to convert a legacy software into another programming language. Finally, the task of *Code Refinement* requires to automatically patch a bugged code.

Experiments show that GraphCodeBERT achieve significant results in both source code and natural language representations. In particular, the model achieves highest performances in the task of Code Clone Detection, reaching F1-score of 0.97.

## 5.3 Dataset

Our intent is to assess transformer-based architectures on the task of code smell detection. For this reason, we aim to fine-tune various pre-trained architectures to learn a valid code representation, allowing to infer smells in code snippets. To achieve the goal, we define a comprehensive collection of code smells from four existing datasets, namely *Madeyski et al. (MLCQ)* [84], *Arcelli et al.* [42], *Walter et al.* [122], and *Sharma et al.* [113].

The resulting collection includes 4'901'768 samples from eleven types of code smells: six at the method level and five at the class level. Method-level smells include *Complex Conditional, Complex Method, Feature Envy, Long Method, Long Parameter List, Switch Statements.* Class-level smells comprise *Data Class God Class, Multifaceted Abstraction, Refused Bequest*, and *Shotgun Surgery.* We note that code smells on classes are not evaluated in our experiments. However, these samples are included in the dataset to facilitate our future experiments. The dataset is available to download in zip format [1].

### 5.3.1 Dataset Collection

We download the code samples from the four selected sources. From Arcelli et al.'s dataset, we collect samples for Feature Envy, Long Method, God Class, Data Class, Long Parameter List, and Switch Statements. From Walter et al.'s dataset, we select examples for Refused Bequest and Shotgun Surgery; we exclude samples of other code smell types to avoid duplicates of those extracted from the Arcelli et al.'s dataset. In fact, the two datasets contain code samples deriving from different versions of the same code corpus. From Sharma et al.'s dataset, we extracted samples for Complex Conditional, Complex Method, and Multifaceted Abstraction; we excluded samples of Feature Envy, because re-

[1] *https : //drive.google.com/file/d/1iKJ54ox4GkIaiH9VcqhIinvFdnl6loli/view?usp = drive_link*

searches identify the code smell at the class-level rather than at the method-level. Finally, we collect code samples from Madeyski et al.'s MLCQ for Data Class, God Class, Feature Envy, and Long Method.

Except for the Sharma et al.'s collection, the other datasets consist of lists in CSV format, containing references to code snippets to be downloaded from various sources. Therefore, we reconstruct each dataset separately. For each data source, we divide code smells in different folders, splitting their examples into positive (i.e., smelly codes) and negative (i.e., non-smelly codes) samples. In addition, we convert the labels having values in a severity range into a binary form. Instances having severity value greater than 0 were marked as positive, while those having severity equal to 0 were categorized as negative.

### 5.3.2 Data Pre-processing

To improve the reliability of our experiments, we pre-process data from each source. In MLCQ dataset, each datapoint represents an individual review made by an expert for a specific code snippet. A review comprises various information, such as the type of code smell identified, its severity, and the source code analyzed. As a consequence, MLCQ contains multiple datapoints representing the same code snippet, reviewed by different experts. To avoid biases during the fine-tuning, we removed these duplicate reviews. In addition, we excluded those codes having both positive and negative reviews for a certain code smell (i.e., the same code has been identified as smelly and non-smelly by different experts). Finally, we necessarily skipped samples that are either unavailable for download or not retrieved correctly.

### 5.3.3 Dataset Structure

We merged code samples from all sources into a single comprehensive dataset. The collection is organized in two categories, class-level smells and method-level smells. Each category contains different folders representing types of code smells. Furthermore, the samples are stored in individual files containing code snippets (method or classes), categorized as either a positive or negative examples for a specific code smell. The structure of the dataset proposed is showed in the figure 5.7.

The resulting dataset is summarized in tables 5.2 and 5.3, divided into smell categories. For each code smell, we also provide the number of positive and negative instances.

| *Code Smell* | *Positive Datapoints* | *Negative Datapoints* | *Total* |
|---|---|---|---|
| Complex Method | 45'273 | 2'227'302 | 2'272'575 |
| Complex Conditional | 20'379 | 2'252'196 | 2'272'575 |
| Feature Envy | 95 | 2'147 | 2'242 |
| Long Method | 73 | 213 | 286 |
| Long Parameter List | 36 | 168 | 250 |
| Switch Statements | 82 | 206 | 288 |

**Table 5.2:** Datapoints for method-level code smells

| *Code Smell* | *Positive Datapoints* | *Negative Datapoints* | *Total* |
|---|---|---|---|
| Data Class | 247 | 1'965 | 2'212 |
| God Class | 278 | 1914 | 2'192 |
| Multifaceted Abstraction | 847 | 318'651 | 319'498 |
| Refused Bequest | 148 | 14'697 | 14'845 |
| Shotgun Surgery | 1'084 | 13'767 | 14'851 |

**Table 5.3:** Datapoints for class-level code smells

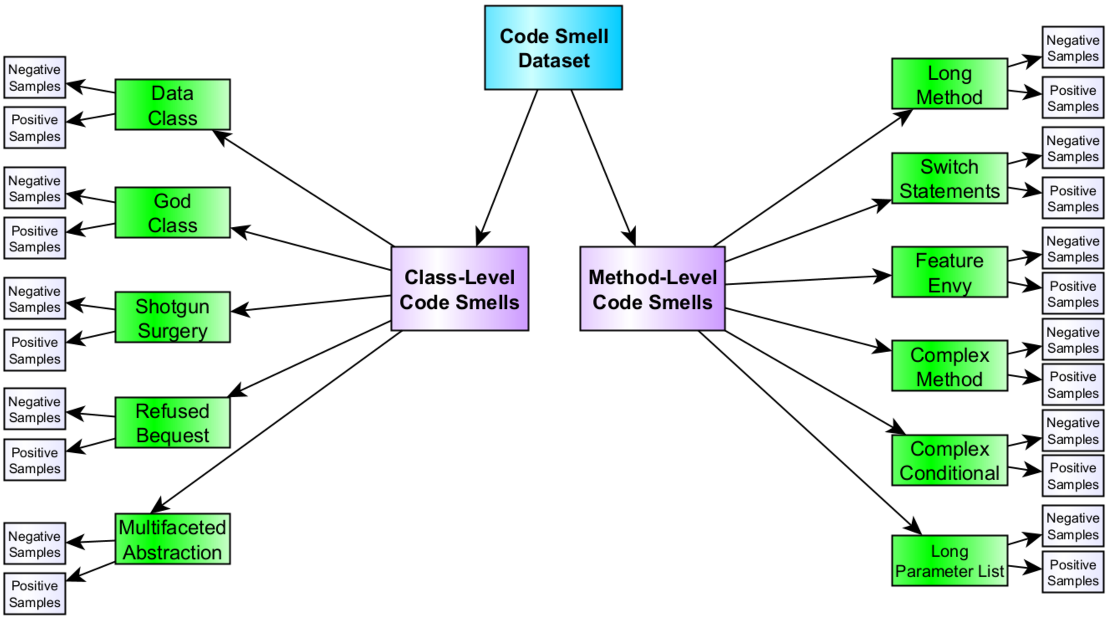

**Figure 5.7:** Structure of the code smell dataset

## 5.4 Experiments Evaluation

The goal of our approach is to assess performance of transformer-based architectures on automated detection of method-level code smells. For this reason, we select four pre-trained transformer models. Three of these architectures are specialized in code representation: CuBERT, CodeBERT, and GraphCodeBERT. We compare their performance with BERT, an architecture pre-trained on general NLP tasks. To ensure a fair comparison with the other models, we use the 'large' configuration of BERT, aligning its architecture with the one used for CuBERT.

While our experiments focuses on automated code smell detection at the method-level, the proposed approach can be extended also to class-level smells. The evaluation of code smells on classes involves complex computations that will be addressed in future work.

### 5.4.1 Problem Definition

We model the automated detection of each code smell as a sequence classification problem. The goal of this task is to learn a classifier $C$, a function that assigns a sequence $s$ to a label l from a predefined set of labels $L$ [128]. The sequence classification task is used across many fields. For instance, an interesting application of sequence classification is categorizing documents by topic.

As showed in [122], there exist several relationships between different code smells. These connections can introduce bias into a model's training when it tries to learn the classification of multiple types of code smells simultaneously. For this reason, we aim to detect various smells separately. We define our problem as a set of binary classification tasks, each one focused on a specific method-level code smell. In detail, we define a set of N individual sequence classification tasks $S = (t_1, t_2 ..., t_N)$, where N is the total number of code smell types. Each task $t_i \in S$ $(1 \leqslant i \leqslant N)$ corresponds to the automated detection problem for a specific code smell. For each $t_i$, we define a set of class labels $L_i = [''non\ smelly'', ''smelly'']$. The "non smelly" class label represents codes not impacted by the analyzed code smell (negative samples). The "smelly" label indicates codes affected by the evaluated code smell (positive samples).

The goal of $t_i$ is to learn a classifier function $C_i$, assigning each code sequence $s$ to a class label in $L_i$. A code sequence is the minimal code snippet representing an entire method. Formally, the code smell classifier $C_i$ is a function defined as follows.

$$\forall t_i \in S \quad \exists\, C_i : s \implies l, \quad l \in L_i \tag{5.4}$$

In order to train a classifier to identify smelly code sequences, we first need to convert each input into an embedding vector, capturing the relevant aspects of the code. We create these embeddings by feeding the code sequences into a transformer-based architecture. The detection is then performed by a classification component, added on top of the transformer.

### 5.4.2 Architecture Definition

We want to understand how models pre-trained on code features perform on code smell detection compared to BERT, pre-trained only on NLP tasks. To ensure a fair comparison, we build similar setups for all models used in our experiments. We define an architecture

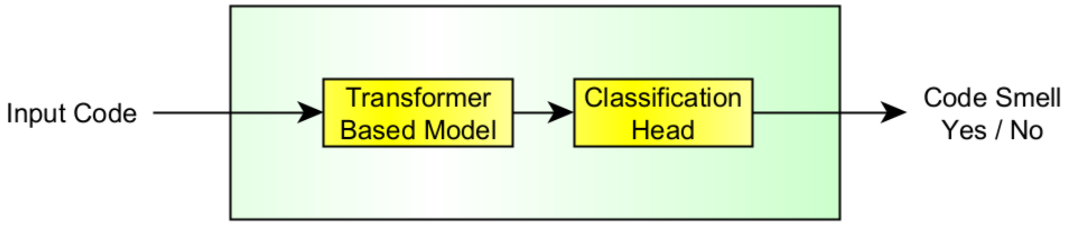

**Figure 5.8:** Architecture proposed for code smell detection

composed by a pre-trained large language model followed by a classification head. We replicate this architecture for each pre-trained model considered. As a result, we obtain four different configurations to use in our experiments: *BERT-Classifier, CuBERT-Classifier, CodeBERT-Classifier*, and *GraphCodeBERT-Classifier*. Every model is fine-tuned on each individual code smell type. In the following, we describe the configurations used during the experiments.

**BERT-Classifier** We use BERT Tokenizer from Hugging Face library to pre-process method codes in input. The resulting embeddings are fed into BERT Large to learn a representation of smelly codes. The vector generated is then forwarded to a dropout layer, to reduce over-fitting during the fine-tuning process. Then, the information is fed into a linear layer. The output of BERT-Classifier represents the binary prediction for the method analyzed.

**CuBERT-Classifier** We use CuBERT Tokenizer for encoding the program methods. The tokenized sequences are passed to CuBERT, that learns embeddings for methods in input. These embedding vectors are then fed into a dropout layer and a linear layer. The output of CuBERT-Classifier indicates whether the method in input is impacted by a specific code smell.

**CodeBERT-Classifier** Following the GitHub's implementation of CodeBERT, we embed input sequences with Hugging Face's Tokenizer for RoBERTa. The encoding is then fed into CodeBERT. The embedding produced by CodeBERT is forwarded to the classification head. This is composed by a linear layer, followed by a dropout and another linear layer. The model outputs a binary classification for the input code sequence. This indicates whether a particular code smell is present in the input method.

**GraphCodeBERT-Classifier** We use the same approach proposed by the CodeBERT's repository on GitHub. The input methods are tokenized with RoBERTa Tokenizer from Hugging Face. The input embedding is passed to GraphCodeBERT to learn the representation of code smells. The embedding generated is then forwarded to the classification component. This includes three layers: linear, dropout, and another linear. The output prediction represents the presence or absence of a specific code smell in the method analyzed.

### 5.4.3 Experiments

To evaluate model performance under different conditions, we design two experiments for each code smell. In the first experiment, models are fine-tuned and tested on a *balanced dataset*, ensuring equal number of positive and negative samples. For each code smell, we identify the class with fewer samples (positive or negative) and randomly select the same number of samples from the majority class. In the second experiment we fine-tune and evaluate models on an *imbalanced dataset*. It contains all available samples for a specific code smell, maintaining the ratio between positive and negative instances. This experiment simulates a real-world scenario where code smells affect only a small portion of the codebase.

In both experiments, samples are randomly shuffled. Then, they are split into 80% training, 10% validation and 10% test sets. This procedure is required to minimize potential biases deriving from arrangements of datapoints in the splits. The resulting sets are stored in JSONL format. Each line represents a datapoint composed by a method to classify and the corresponding classification label. Each model is fine-tuned for four epochs in a supervised approach, using training and validation sets.

Models' performance is then evaluated on test sets using a comprehensive set of metrics: accuracy, precision, recall, F1-score, and AUC. In the following we detail the performance achieved by the models on test sets for each code smell.

#### Complex Method

All models showed significant efficiency in detecting Complex Method's code smell. These positive results might be attributed to two key factors. First, the large number of positive samples could have aided the transformers in learning an accurate representation of affected methods. Second, Complex Method might be intrinsically easier to identify compared to more subtle code smells, because of its evident characteristics. The following tables (5.4, 5.5) summarize the overall performance achieved in the two experiments. For the imbalanced dataset, the highest accuracy is 0.99 and is achieved equally by CuBERT-Classifier, CodeBERT-Classifier, and BERT-Classifier. However, CodeBERT-Classifier achieved high performance in terms of F1-score (0.70) and AUC (0.82), demonstrating its effectiveness in identifying Complex Methods in imbalanced datasets. For the balanced dataset, instead, CuBERT-Classifier outperforms other models, achieving accuracy, F1-score, and AUC equal to 0.98.

| *Complex Method - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.99 | 0.84 | 0.58 | 0.68 | 0.79 |
| CodeBERT-Classifier | 0.99 | 0.76 | 0.65 | 0.70 | 0.82 |
| GraphCodeBERT-Classifier | 0.98 | 0.82 | 0.28 | 0.42 | 0.64 |
| BERT-Classifier | 0.99 | 0.68 | 0.45 | 0.57 | 0.72 |

**Table 5.4:** Complex Method. Metrics for the imbalanced dataset (All available samples)

| *Complex Method - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.98 | 0.96 | 0.99 | 0.98 | 0.98 |
| CodeBERT-Classifier | 0.95 | 0.94 | 0.97 | 0.95 | 0.96 |
| GraphCodeBERT-Classifier | 0.97 | 0.95 | 0.99 | 0.97 | 0.97 |
| BERT-Classifier | 0.95 | 0.92 | 0.97 | 0.95 | 0.95 |

**Table 5.5:** Complex Method. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### Complex Conditional

When detecting Complex Conditionals, models pre-trained on code achieved better performance compared to BERT. This is particularly evident in the imbalanced dataset, where BERT-Classifier cannot identify Complex Conditionals. In the imbalanced experiment GraphCodeBERT-Classifier achieves AUC of 0.97, while having very low precision and F1-score. This could derive from high imbalance of the dataset. As a consequence, the model is correctly identifying smelly methods, while misclassifying a great number of negative samples. However, CuBERT-Classifier and CodeBERT-Classifier still achieve significant results in the imbalanced dataset.

All models significantly improve their classifications for Complex Conditionals on the balanced dataset compared to the imbalanced one. While BERT-Classifier reaches an accuracy of 0.95, the other models achieve higher scores. CuBERT-Classifier achieves an accuracy of 0.99, while CodeBERT-Classifier and GraphCodeBERT-Classifier both reach accuracy equal to 0.97. The performance is summarized in tables 5.6, 5.7.

| *Complex Conditional - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.98 | 0.97 | 0.79 | 0.87 | 0.89 |
| CodeBERT-Classifier | 0.99 | 0.92 | 0.68 | 0.78 | 0.84 |
| GraphCodeBERT-Classifier | 0.93 | 0.12 | 0.99 | 0.21 | 0.97 |
| BERT-Classifier | — | — | — | — | — |

**Table 5.6:** Complex Conditional. Metrics for the imbalanced dataset (All available samples)

| *Complex Conditional - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.99 | 0.97 | 0.79 | 0.87 | 0.89 |
| CodeBERT-Classifier | 0.97 | 0.95 | 0.99 | 0.97 | 0.97 |
| GraphCodeBERT-Classifier | 0.97 | 0.95 | 0.99 | 0.97 | 0.97 |
| BERT-Classifier | 0.95 | 0.94 | 0.96 | 0.95 | 0.95 |

**Table 5.7:** Complex Conditional. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### Feature Envy

CodeBERT-Classifier reached the highest AUC (0.75) in Feature Envy detection in the imbalanced experiment. The other models experienced significant difficulty in identifying this code smell, reaching a maximum AUC of 0.59 with BERT-Classifier. These results might be caused by the proportion of code samples, which is heavily skewed towards non-smelly instances. In the balanced experiments, all models achieves higher performance. In particular, CuBERT-Classifier demonstrates an AUC, F1-score, and accuracy equal of 0.89. The overall performance achieved in the experiments is presented in tables 5.8, 5.9.

| *Feature Envy - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.95 | 0.25 | 0.10 | 0.14 | 0.54 |
| CodeBERT-Classifier | 0.75 | 0.67 | 1.0 | 0.80 | 0.75 |
| GraphCodeBERT-Classifier | 0.96 | 1.0 | 0.11 | 0.2 | 0.56 |
| BERT-Classifier | 0.94 | 0.29 | 0.20 | 0.23 | 0.59 |

**Table 5.8:** Feature Envy. Metrics for the Imbalanced Dataset (All available samples)

| *Feature Envy - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.89 | 0.89 | 0.89 | 0.89 | 0.89 |
| CodeBERT-Classifier | 0.65 | 0.59 | 1.0 | 0.74 | 0.65 |
| GraphCodeBERT-Classifier | 0.75 | 0.69 | 0.9 | 0.78 | 0.75 |
| BERT-Classifier | 0.83 | 0.75 | 1.0 | 0.86 | 0.83 |

**Table 5.9:** Feature Envy. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### Long Method

Tables 5.10, 5.11 show the results of the architectures in detecting Long Methods in imbalanced and balanced scenarios. In these experiments, CuBERT-Classifier outperforms other models. This architecture achieved F1-score of 0.93 and AUC of 0.98 in the imbalanced experiment. For the balanced scenario, CuBERT-Classifier achieved F1-score of

0.94 and AUC of 0.94. On the contrary, in the imbalanced scenario CodeBERT-Classifier does not achieve satisfactory performance. This could be attributed to two factors. First, the restricted number of samples could limit the model's ability to learn the representation and classification of Long Methods. Second, this code smell could be more subtle to be identified with respect to others.

| *Long Method - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.96 | 0.87 | 1.0 | 0.93 | 0.98 |
| CodeBERT-Classifier | — | — | — | — | — |
| GraphCodeBERT-Classifier | 0.86 | 1.0 | 0.2 | 0.33 | 0.6 |
| BERT-Classifier | 0.86 | 0.71 | 0.71 | 0.71 | 0.81 |

**Table 5.10:** Long Method. Metrics for the imbalanced dataset (All available samples)

| *Long Method - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.94 | 0.89 | 1.0 | 0.94 | 0.94 |
| CodeBERT-Classifier | 0.57 | 0.54 | 1.0 | 0.70 | 0.57 |
| GraphCodeBERT-Classifier | 0.78 | 0.70 | 1.0 | 0.82 | 0.78 |
| BERT-Classifier | 0.75 | 0.67 | 1.0 | 0.80 | 0.75 |

**Table 5.11:** Long Method. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### Long Parameter List

The task of detecting Long Parameter Lists is challenging due to the scarcity of samples, in particular the limited number of positive instances. However, the majority of model configurations are still able to learn method's representation and classifying them. Models demonstrate different performance in detecting Long Parameter Lists on imbalanced and balanced experiments. In the first case, only CuBERT-Classifier and CodeBERT-Classifier are able to classify methods in input. Specifically, CuBERT-Classifier achieves AUC of 0.72, F1-score of 0.57 and accuracy equal to 0.86. For the balanced scenario, instead, CuBERT-Classifier, GraphCodeBERT-Classifier, and BERT-Classifier are able to identify Long Parameter Lists in methods being analyzed. In this scenario, BERT-Classifier achieves significant AUC (0.83), F1-score (0.86) and accuracy (0.83). Results are reported in tables 5.12, 5.13.

| *Long Parameter List - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.86 | 0.67 | 0.5 | 0.57 | 0.72 |
| CodeBERT-Classifier | 0.5 | 0.5 | 1.0 | 0.67 | 0.5 |
| GraphCodeBERT-Classifier | — | — | — | — | — |
| BERT-Classifier | — | — | — | — | — |

**Table 5.12:** Long Parameter List. Metrics for the imbalanced dataset (All available samples)

| *Long Parameter List - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.67 | 0.60 | 1.0 | 0.75 | 0.67 |
| CodeBERT-Classifier | — | — | — | — | — |
| GraphCodeBERT-Classifier | 0.63 | 0.60 | 0.75 | 0.67 | 0.63 |
| BERT-Classifier | 0.83 | 0.75 | 1.0 | 0.86 | 0.83 |

**Table 5.13:** Long Parameter List. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### Switch Statements

Tables 5.14 and 5.15 summarize performance of the models in detecting Switch Statements. Due to the limited number of samples available also during fine-tuning, three models cannot complete the task effectively on the imbalanced dataset. However, CuBERT-Classifier successfully handles this task even on imbalanced scenario, achieving AUC of 0.82, accuracy of 0.79 and F1-score of 0.75. For the balanced dataset, all models reached significant performance in the test set. In particular, all models pre-trained on code achieved a maximum score correctly detecting all samples being evaluated. However, BERT-Classifier achieved lower results: AUC of 0.75, F1-score of 0.80 and accuracy of 0.75.

| *Switch Statements - Imbalanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 0.79 | 0.64 | 0.90 | 0.75 | 0.82 |
| CodeBERT-Classifier | — | — | — | — | — |
| GraphCodeBERT-Classifier | — | — | — | — | — |
| BERT-Classifier | — | — | — | — | — |

**Table 5.14:** Switch Statements. Metrics for the imbalanced dataset (All available samples)

| *Switch Statements - Balanced Dataset* | | | | | |
|---|---|---|---|---|---|
| *Model Configuration* | *Accuracy* | *Precision* | *Recall* | *F1-score* | *AUC* |
| CuBERT-Classifier | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| CodeBERT-Classifier | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| GraphCodeBERT-Classifier | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| BERT-Classifier | 0.75 | 0.67 | 1.0 | 0.80 | 0.75 |

**Table 5.15:** Switch Statements. Metrics for the balanced dataset (50% positive - 50% negative datapoints)

### 5.4.4 Discussion

Our study investigates code smell detection using four transformer-based architectures: BERT, CuBERT, GraphCodeBERT, and CodeBERT. Similarly to the approach presented in [71], we have used these models for constructing code embeddings. However, our approach proposes also to fine-tune the transformers for learning representation of smelly code. Specifically, we add a classification head on top of the transformers and we fine-tune the entire configuration on each code smell.

The fine-tuning process allows to create effective embeddings of methods affected by code smells. In particular, our approach outperforms the one proposed in [71] for detecting Long Methods with CuBERT embeddings. While the technique proposed in the paper achieves F1-score of 0.75, our approach reaches F1-score of 0.93/0.94.

We noticed that the ratio of positive/negative examples and the type of code smell significantly impacted results of the experiments.

In Complex Method experiments, all model configurations achieved high AUC with both balanced and imbalanced datasets. Similar results are achieved with the detection of Complex Conditional. However, in this case BERT-Classifier is unable to successfully complete the task on the imbalanced dataset.

In Feature Envy experiments, only CodeBERT-Classifier achieved relevant AUC (0.75) with the imbalanced dataset. However, the overall performance is increased with the balanced dataset; the results ranges between a minimum AUC of 0.65 (CodeBERT-Classifier), and a maximum AUC of 0.89 (CuBERT-Classifier).

While all architectures successfully detected Long Methods in the balanced dataset, CodeBERT-Classifier failed in the imbalanced scenario. In both cases, CuBERT-Classifier reached the highest value of AUC (0.98 in the imbalanced experiment, 0.94 in the balanced one).

Finally, the detection of Long Parameter List and Switch Statements was particularly challenging. In the first case, the imbalanced experiment was completed only by CuBERT-Classifier and CodeBERT-Classifier. In the latter case, only CuBERT-Classifier completed the imbalanced experiment. The overall performance is increased in the balanced experiments. BERT-Classifier achieved the highest AUC of 0.83 for Feature Envy detection, while the other three architectures correctly detected Switch Statements (AUC of 1.0).

These experiments showed that models pre-trained on code representation generally achieve better results with respect to the one not pre-trained on code. This is particularly evident in imbalanced datasets, where BERT cannot always accomplish the task requested.

Furthermore, the size of the fine-tuning dataset could also influence the detection performance. Current experiments on Feature Envy, Long Method, Long Parameter List, and Switch Statements detection might be limited by the small amount of fine-tuning examples. Therefore, in the future we aim to collect larger datasets for our experiments. These could enhance the performance and increase the generalizability of new architectures.

Alternatively, we could explore different learning methods. The zero-shot learning is a valid option for our context. This technique allows to classify elements of the test dataset that were not analyzed during the training phase. Therefore, zero-shot learning could be useful for improving the detection of code smells having only few labeled examples available.

# Chapter 6

# Conclusions and Future Work

The study presented in the thesis aims to find alternative solutions for solving well-known tasks related to code analysis. The research is focused on automated approaches, applying various neural language models on source and binary code analysis.

Chapter 4 introduces the first contribution of the thesis, a benchmark for binary function representation. We developed a large dataset, consisting of binary files and corresponding assembly functions. The benchmark included five heterogeneous tasks on binary functions, namely binary similarity, function search, signature recovery, function naming, and compiler provenance. For each task, we provided a set of datapoints to be evaluated both for training and testing DNNs. The benchmark provides not only the samples and the datapoints to be inferred, but also a useful tool for evaluating the models' solutions. The benchmark can be used to compare architectures both offline, using a script provided with the dataset, and online, using the EvalAI platform.

With this contribution we hope to encourage the development of new state-of-the-art models for binary code representation. These tools are fundamental for understanding features of executable files when source code is not available, for example in case of malware analysis or for detecting plagiarism in software.

As future work, we aim to improve baseline scores provided for the tasks of function search, function naming, and signature recovery. For the task of function search, we propose to fine-tune the baseline model, SAFE, on our dataset. This could improve the results achieved in detecting the K most similar functions. Similarly, we intend to fine-tune the pre-trained baseline of function naming on our dataset. This approach could increase the capabilities of the model in inferring names of the evaluated binary functions. Another option is to train and test different DNNs on BinBench. The goal would be to identify the DNN that performs best, and use it as baseline for the function naming task.

For the task of signature recovery, we used as baseline a transformer trained from scratch. Therefore, to improve the baseline score we can consider two alternatives. The first is to re-train the transformer, using different configurations and a larger training dataset. The second option is to use a different architecture as baseline. For example, we could fine-tune a LLM trained on code, such as CuBERT or CodeBERT, on our task.

Furthermore, in the future we aim to introduce new tasks for addressing other relevant scenarios. In particular, we are interested in extending the benchmark to tasks defined over entire binary codes. Another important extension is increasing the size of the dataset. This can be achieved using additional open-source packages and compiling the source codes over various architectures. These aspects could facilitate models in improving their generalization capabilities.

Chapter 5 describes the second contribution of the thesis, an automated approach for code smell detection based on large language models. We collected a comprehensive dataset, containing different types of code smells. The research evaluates four different transformer-based architectures on the task of automated code smell detection. These LLMs are pre-trained on different tasks. More specifically, three of them are pre-trained on tasks related to code analysis, while the fourth architecture is pre-trained on NLP tasks. We develop four classifiers, one for each LLM. These architectures are composed by two parts, the LLM and a classifier head. We fine-tuned each model for detecting code smells as a form of binary classification.

The study is a starting point for improving the detection of code smells, a fundamental analysis facilitating the code refactoring. Results are promising, in particular for classifiers that incorporate LLMs pre-trained on code snippets.

For future work, we aim to evaluate other models on the task of code smell detection. Furthermore, to generalize our automated approach, we intend to train models on a larger dataset. Detection performance generally improves with more training samples, and this could be crucial for detecting less common smells. We plan to collect additional examples for the code smells we have already examined. Furthermore, we aim to expand our research on other types of code smells at method-level.

We are also interested in extending our study to the detection of code smells affecting entire classes. By detecting both method-level and class-level smells, we could gain a broad comprehension of design issues affecting the source code.

We are furthermore interested in an in-depth investigation of relationships between smells and code vulnerabilities. To this end, we intend to train LLMs for recognizing code smells that could be used to exploit specific vulnerability patterns.

Finally, we aim to use the resulting knowledge to investigate the presence of code smells in binary codes. This research could allow to understand whether these wrong design choices are still evident in the executable files.

# Appendix A

# Automata

The section provides a description of the automata mentioned in chapter 3.1.1. Automata are abstract machines with a central unit, temporary storage, and the ability to process input strings and produce corresponding output. Automata can be deterministic or non deterministic. In the first case, for each state and input provided, there exists only one possible next state to transition to. In the second case, instead, a given state and an input symbol can potentially lead to different transitions towards multiple next states.

## A.1 Finite State Automata

A *finite state automaton* is mathematical model of computation, having internal states transition rules, inputs and outputs. Formally, the finite state automaton is defined as follows.

$$M = (Q, \Sigma, \delta, q_0, F)$$

Where:
- Q represents the finite set of internal states,
- $\Sigma$ is the input alphabet,
- $\delta : Q \times \Sigma \rightarrow$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ represents the set of final states.

This automaton has no temporary storage. It can only maintain limited amount of data in a finite number of states. The transition between one internal state and another is defined by the transition function. These finite state automata are represented through state transition diagrams. These graphs show how the input is consumed, one symbol at a time. The automata progress through different states depending on the symbol analyzed. An input string is accepted only if the finite automata is in a final state when reading the last symbol of the string [80]. Otherwise, the string is rejected.

A language $L$ accepted by a state transition automaton $M$ is composed by all strings on the input alphabet $\Sigma$ accepted by $M$. In particular, the language $L$ is *regular* if and only if there exists a finite state automata $M$ that recognize the language L.

An example of regular language recognized by a finite state automaton is the following.

$$L = \{aSb : S \in \{a,b\}^*\}$$

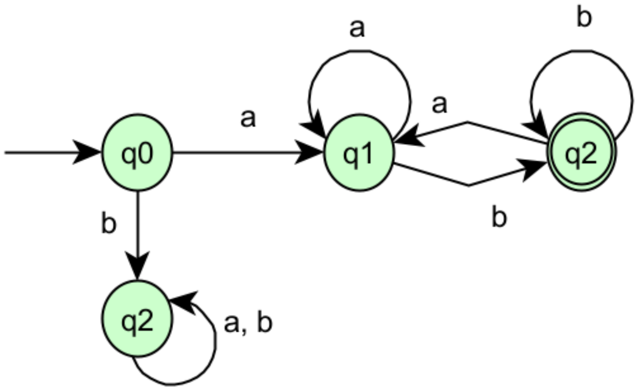

**Figure A.1:** State transition graph representing the finite state automata M, accepting the language $L = \{aSb : S \in \{a,b\}^*\}$

## A.2 Pushdown Automata

Similarly to finite state automata, a *non deterministic pushdown automaton* is mathematical model of computation used to recognize context-free languages. Due to the nature of these languages, pushdown automata require unbounded memory that is implemented as a stack [80].
Formally, a pushdown automata is:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Where:
- Q represents the finite set of internal states,
- $\Sigma$ is the input alphabet,
- $\Gamma$ is the stack alphabet,
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Sigma \rightarrow set_{finite}(Q \times \Gamma^*)$ is the transition function, in which $\epsilon$ represents the empty string,
- $q_0 \in Q$ is the initial state,
- $z_0 \in \Gamma$ is the starting symbol for the stack,
- $F \subseteq Q$ represents the set of final states.

For each context-free language exists a non deterministic pushdown automaton that recognize it.

## A.3 Turing Machine

A *Turing machine* is a theoretical model operating on symbols stored in an unbounded tape and defined by an alphabet. The tape is composed of cells, each capable of storing a single symbol [80]. The Turing Machine operates on these symbols according to defined rules. Formally, the machine is defines as follows.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$$

Where:
- Q represents the finite set of internal states of the Turing machine,
- $\Sigma \subseteq \Gamma - \Box$ indicates the input alphabet,
- $\Gamma$ is the tape alphabet,
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the initial state of the Turing machine,
- $\Box \in \Gamma$ is the blank symbol,
- $F \subseteq Q$ is the set of final states of the machine.

Given the current state and the current symbol analyzed, the transition function $\delta$ returns a specific output. This consists of a new state to move toward, a new symbol to overwrite the current one on the tape, and a direction (left or right) for the central unit to move toward.

A language is categorized as recursively enumerable if there exists a Turing machine that can enumerate all legal strings of that language. The language is also defined recursively enumerable if there is a Turing machine that accepts it.

## A.4 Linear Bounded Automata

A *linear bounded automaton* is non deterministic Turing machine, having size of the tape linearly proportional to the size of its input. A linear bounded automaton is defined as follows.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$$

Where:
- Q is the finite set of internal states of the automaton,
- $\Sigma \subseteq \Gamma - \Box$ represents the input alphabet,
- $\Gamma$ indicates the tape alphabet,
- $\delta : Q \times \Sigma \to 2^{Q \times \Sigma \times \{L,R\}}$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $\Box \in \Gamma$ is the blank symbol,
- $F \subseteq Q$ represents the set of final states of the automaton.

The input alphabet $\Sigma$ has to include the two symbols: [ , ]. The transition function $\delta$ takes as input the current state and the current symbol analyzed. The output of $\delta$ involves three elements: a new state, a symbol to substitute the current symbol on the tape, and a direction (left or right) for the central unit to move. Since the automaton is non deterministic, there exists a set of transitions that can be chosen as next.

For each context-sensitive language not including the empty string $\epsilon$, there exists a linear bounded automata allowing to recognize it [80].

# Bibliography

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018.

[4] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[6] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12, 03 2015.

[7] E. Alsentzer, J. Murphy, W. Boag, W.-H. Weng, D. Jindi, T. Naumann, and M. McDermott. Publicly available clinical BERT embeddings. In A. Rumshisky, K. Roberts, S. Bethard, and T. Naumann, editors, *Proceedings of the 2nd Clinical Natural Language Processing Workshop*, pages 72–78, Minneapolis, Minnesota, USA, June 2019. Association for Computational Linguistics.

[8] W. Antoun, F. Baly, and H. Hajj. Arabert: Transformer-based model for arabic language understanding. *arXiv preprint arXiv:2003.00104*, 2020.

[9] F. Artuso, G. A. D. Luna, L. Massarelli, and L. Querzoni. In nomine function: Naming functions in stripped binaries with neural networks, 2021.

[10] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.*, 108:115–138, 2019.

[11] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[12] M. Ben-Ari. Understanding programming languages. 1996.

[13] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, 2003.

[14] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[15] P. F. Brown, V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer. Class-based *n*-gram models of natural language. *Computational Linguistics*, 18(4):467–480, 1992.

[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[17] A. Caliskan, F. Yamaguchi, E. Dauber, R. E. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[18] J. M. Chambers. Object-Oriented Programming, Functional Programming and R. *Statistical Science*, 29(2):167 – 180, 2014.

[19] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, jan 2024. Just Accepted.

[20] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao. *HIMALIA: Recovering Compiler Optimization Levels from Binaries by Deep Learning: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 1*, pages 35–47. 01 2019.

[21] N. Chomsky. On certain formal properties of grammars. *Inf. Control.*, 2(2):137–167, 1959.

[22] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 99–116, USA, 2017. USENIX Association.

[23] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw. Pract. Exp.*, 25(7):811–829, 1995.

[24] C. Clark, K. Lee, M. Chang, T. Kwiatkowski, M. Collins, and K. Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 2924–2936. Association for Computational Linguistics, 2019.

[25] F. Console, G. D'Aquanno, G. A. D. Luna, and L. Querzoni. Binbench: a benchmark for x64 portable operating system interface binary function representations. *PeerJ Comput. Sci.*, 9:e1286, 2023.

[26] F. Console, L. Querzoni, G. D. Luna, and G. D'Aquanno. Binbench challenge on eval.ai. `https://eval.ai/web/challenges/challenge-page/1980/overview`, 2023. Accessed: (07/04/23).

[27] F. Console, L. Querzoni, G. D. Luna, and G. D'Aquanno. Binbench: Dataset for binary function representations. `https://figshare.com/articles/dataset/BinBench_Dataset_for_Binary_Function_Representations/21546111/2`, 2023. Accessed: (07/04/23).

[28] Y. David, U. Alon, and E. Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.*, 4(OOPSLA):225:1–225:28, 2020.

[29] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. *SIGPLAN Not.*, 51(6):266–280, jun 2016.

[30] Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language*

*Design and Implementation*, PLDI 2017, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Y. David and E. Yahav. Tracelet-based code search in executables. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[32] W. De Vries, A. van Cranenburgh, A. Bisazza, T. Caselli, G. van Noord, and M. Nissim. Bertje: A dutch bert model. *arXiv preprint arXiv:1912.09582*, 2019.

[33] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[34] J. DeYoung, S. Jain, N. F. Rajani, E. Lehman, C. Xiong, R. Socher, and B. C. Wallace. ERASER: A benchmark to evaluate rationalized NLP models. *CoRR*, abs/1911.03429, 2019.

[35] S. H. Ding, B. C. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[36] W. B. Dolan and C. Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing, IWP@IJCNLP 2005, Jeju Island, Korea, October 2005, 2005*. Asian Federation of Natural Language Processing, 2005.

[37] L. Dong, S. Xu, and B. Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[38] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5, 01 2005.

[39] J. Escalada, F. Ortin, and T. Scully. An efficient platform for the automatic extraction of patterns in native code. *Sci. Program.*, 2017:3273891:1–3273891:16, 2017.

[40] J. Escalada, T. Scully, and F. Ortin. Improving type information inferred by decompilers with supervised machine learning. *CoRR*, abs/2101.08116, 2021.

[41] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[42] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empir. Softw. Eng.*, 21(3):1143–1191, 2016.

[43] M. Fowler. Refactoring: Improving the design of existing code. In D. Wells and L. A. Williams, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer, 2002.

[44] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 3703–3714, 2019.

[45] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[46] H. Gao, S. Cheng, Y. Xue, and W. Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 607–619, New York, NY, USA, 2021. Association for Computing Machinery.

[47] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12:2451–2471, 2000.

[48] H. Gholamalinezhad and H. Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, abs/2009.07485, 2020.

[49] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[50] D. Grune, K. v. Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen. *Modern Compiler Design.* Springer Publishing Company, Incorporated, 2nd edition, 2012.

[51] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin,

D. Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[52] M. Hadj-Kacem and N. Bouassida. A hybrid approach to detect code smells using deep learning. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018.

[53] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, Z. Yang, Y. Zhang, and D. Tao. A survey on vision transformer. *IEEE Trans. Pattern Anal. Mach. Intell.*, 45(1):87–110, 2023.

[54] I. Haq and J. Caballero. A survey of binary code similarity. *ACM Computing Surveys (CSUR)*, 54:1 – 38, 2021.

[55] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1667–1680, New York, NY, USA, 2018. Association for Computing Machinery.

[56] S. Heath. 1 - what is an embedded system? In S. Heath, editor, *Embedded Systems Design (Second Edition)*, pages 1–14. Newnes, Oxford, second edition edition, 2002.

[57] C. V. Hegde and S. Patil. Unsupervised paraphrase generation using pre-trained language models. *CoRR*, abs/2006.05477, 2020.

[58] F. Herrera, F. Charte, A. Rivera Rivas, and M. J. Del Jesus. *Multilabel Classification.* 01 2016.

[59] A. Ho, A. M. T. Bui, P. T. Nguyen, and A. D. Salle. Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE 2023, Oulu, Finland, June 14-16, 2023*, pages 229–234. ACM, 2023.

[60] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[61] S. ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4):185–196, 1993.

[62] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[63] D. Jurafsky and J. H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition,*

*2nd Edition.* Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

[64] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 2020.

[65] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356, 2018.

[66] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav. Towards neural decompilation. *CoRR*, abs/1905.08325, 2019.

[67] S. Khanuja, S. Dandapat, A. Srinivasan, S. Sitaram, and M. Choudhury. GLUECoS: An evaluation benchmark for code-switched NLP. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3575–3585, Online, July 2020. Association for Computational Linguistics.

[68] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338, 2013.

[69] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *arXiv preprint arXiv:2011.10749*, 2020.

[70] R. Kirkov and G. Agre. Source code analysis–an overview. *Cybernetics and Information Technologies*, 10(2):60–77, 2010.

[71] A. Kovacevic, J. Slivka, D. Vidakovic, K. Grujic, N. Luburic, S. Prokic, and G. Sladic. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Syst. Appl.*, 204:117607, 2022.

[72] N. Krishnamoorthy, S. K. Debray, and K. Fligg. Static detection of disassembly errors. In A. Zaidman, G. Antoniol, and S. Ducasse, editors, *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pages 259–268. IEEE Computer Society, 2009.

[73] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 255–270. USENIX, 2004.

[74] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, New York, NY, USA, 2013. Association for Computing Machinery.

[75] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.

[76] E. Lehman, J. DeYoung, R. Barzilay, and B. C. Wallace. Inferring which medical treatments work from reports of clinical trials. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3705–3717, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[77] X. Li, Q. Yu, and H. Yin. Palmtree: Learning an assembly language model for instruction embedding. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[78] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph structured objects. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3835–3845. PMLR, 2019.

[79] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022.

[80] P. Linz. *An Introduction to Formal Languages and Automata, Fifth Edition.* Jones and Bartlett Publishers, Inc., USA, 5th edition, 2011.

[81] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou. *$\alpha$ Diff: Cross-Version Binary Code Similarity Detection with DNN*, page 667–678. Association for Computing Machinery, New York, NY, USA, 2018.

[82] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang. Deep learning based code smell detection. *IEEE Trans. Software Eng.*, 47(9):1811–1837, 2021.

[83] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[84] L. Madeyski and T. Lewowski. Mlcq: Industry-relevant code smell data set. *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020.

[85] R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 701–704. IEEE Computer Society, 2005.

[86] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019.

[87] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni. SAFE: self-attentive function embeddings for binary similarity. In R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 309–329. Springer, 2019.

[88] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In Y. Bengio and Y. LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[89] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, 2011.

[90] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[91] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[92] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[93] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

[94] J. Patrick-Evans, L. Cavallaro, and J. Kinder. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, ACSAC '20, page 373–385, New York, NY, USA, 2020. Association for Computing Machinery.

[95] J. Patrick-Evans, M. Dannehl, and J. Kinder. XFL: extreme function labeling. *CoRR*, abs/2107.13404, 2021.

[96] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In Y. Guéhéneuc, F. Khomh, and F. Sarro, editors, *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 93–104. IEEE / ACM, 2019.

[97] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. Learning approximate execution semantics from traces for binary function similarity. *IEEE Trans. Software Eng.*, 49(4):2776–2790, 2023.

[98] F. Petroni, A. Piktus, A. Fan, P. Lewis, M. Yazdani, N. De Cao, J. Thorne, Y. Jernite, V. Karpukhin, J. Maillard, V. Plachouras, T. Rocktäschel, and S. Riedel. KILT: a benchmark for knowledge intensive language tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2523–2544, Online, June 2021. Association for Computational Linguistics.

[99] M. Polignano, P. Basile, M. De Gemmis, G. Semeraro, V. Basile, et al. Alberto: Italian bert language understanding model for nlp challenging tasks based on tweets. In *CEUR workshop proceedings*, volume 2481, pages 1–6. CEUR, 2019.

[100] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, editors, *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, pages 443–456. ACM, 2023.

[101] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[102] A. Rahimian, P. Shirani, S. Alrabaee, L. Wang, and M. Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14, Supplement 1:S146 – S155, 08 2015.

[103] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical*

*Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, Nov. 2016. Association for Computational Linguistics.

[104] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo. Struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 385–394, New York, NY, USA, 2017. Association for Computing Machinery.

[105] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 100–110, New York, NY, USA, 2011. Association for Computing Machinery.

[106] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, page 21–28, New York, NY, USA, 2010. Association for Computing Machinery.

[107] P. V. Roy. Programming paradigms for dummies: what every programmer should know. 2009.

[108] M. Salehie, S. Li, and L. Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 159–168. IEEE Computer Society, 2006.

[109] H. Salehinejad, J. Baarbe, S. Sankar, J. Barfett, E. Colak, and S. Valaee. Recent advances in recurrent neural networks. *CoRR*, abs/1801.01078, 2018.

[110] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation-with applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 74–83, 2008.

[111] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45(11):2673–2681, 1997.

[112] R. W. Sebesta. *Concepts of Programming Languages.* Pearson, 10th edition, 2012.

[113] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis. Code smell detection by deep direct-learning and transfer-learning. *J. Syst. Softw.*, 176:110936, 2021.

[114] T. Sharma, P. Mishra, and R. Tiwari. Designite: a software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, BRIDGE@ICSE 2016, Austin, Texas, USA, May 17, 2016*, pages 1–4. ACM, 2016.

[115] T. Sharma and D. Spinellis. A survey on software smells. *J. Syst. Softw.*, 138:158–173, 2018.

[116] B. Slivnik. Context-sensitive parsing for programming languages. *J. Comput. Lang.*, 73:101172, 2022.

[117] D. Svozil, V. Kvasnicka, and J. Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43–62, 1997.

[118] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345, 2010.

[119] J. Thorne, A. Vlachos, C. Christodoulopoulos, and A. Mittal. FEVER: a large-scale dataset for fact extraction and VERification. In M. Walker, H. Ji, and A. Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 809–819, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[120] S. G. Tzafestas and K. D. Blekas. Hybrid soft computing systems: a critical survey with engineering applications. *Soft Computing in System and Control Technology, Singapore: Word Scientific*, pages 119–168, 1999.

[121] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[122] B. Walter, F. A. Fontana, and V. Ferme. Code smells and their collocations: A large-scale experiment on open-source systems. *J. Syst. Softw.*, 144:1–21, 2018.

[123] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics.

[124] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang. jtrans: jump-aware transformer for binary code similarity detection. In S. Ryu and Y. Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 1–13. ACM, 2022.

[125] J. S. Warford. *Computing fundamentals: The theory and practice of software design with BlackBox Component Builder.* Springer Science & Business Media, 2013.

[126] A. Warstadt, A. Singh, and S. R. Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[127] A. Williams, N. Nangia, and S. R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In M. A. Walker, H. Ji, and A. Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[128] Z. Xing, J. Pei, and E. Keogh. A brief survey on sequence classification. *SIGKDD Explor. Newsl.*, 12(1):40–48, nov 2010.

[129] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct 2017.

[130] D. Yadav, R. Jain, H. Agrawal, P. Chattopadhyay, T. Singh, A. Jain, S. Singh, S. Lee, and D. Batra. Evalai: Towards better evaluation systems for AI agents. *CoRR*, abs/1902.03570, 2019.

[131] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018.

[132] M. Zhang and R. Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, page 91–100, New York, NY, USA, 2015. Association for Computing Machinery.

[133] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu. Delesmell: Code smell detection based on deep learning and latent semantic analysis. *Knowl. Based Syst.*, 255:109737, 2022.