SAPIENZA
UNIVERSITÀ DI ROMA

Deep Learning Based Binary Code Analysis

Facoltà di Ingegneria dell'informazione, informatica e statistica
Ph.D. Program in Engineering in Computer Science

Fiorella Artuso
ID number 1602113

Advisor
Prof. Leonardo Querzoni

Academic Year 2024/2025

Deep Learning Based Binary Code Analysis
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: February 6, 2025

Author's email: artuso@diag.uniroma1.it

Abstract

The exponential growth of software complexity, coupled with the rise of heterogeneous architectures, further complicates the process of manual binary code analysis. Despite its complexity, binary code analysis is extremely valuable, particularly in scenarios where direct access to the source code is unavailable, such as with proprietary software, firmware images, and malware samples. To tackle these challenges, the scientific community has started studying methods for creating automated binary analysis tools based on Deep Learning (DL) that alleviate the workload of human reverse engineers. Unfortunately, there has been a proliferation of such solutions without much effort toward systematization.

This thesis contains three main contributions:

First, we present a comprehensive literature review that spans nine years of research up to 2024. We propose a systematization of 54 research papers, identify a deep learning pipeline common to all these solutions, and provide an in-depth analysis of each of its steps. This analysis highlights key trends across various approaches as well as gaps that need further investigation.

Second, we explore the applicability of Deep Learning solutions to a novel task: the detection of debug information bugs in optimized binaries. This represents a practically important problem, as most software running in production is produced by an optimizing compiler. Current solutions rely on invariants—human-defined rules that embed the desired behavior—whose violation may indicate the presence of a bug. Although this approach has proved effective in discovering several bugs, it is unable to identify bugs that do not trigger invariants. We trained a set of different models borrowed from the NLP community in an unsupervised way on a large dataset of debug traces. Our results show that DNNs are capable of discovering bugs in both synthetic and real datasets. Additionally, with our models we were able to report 12 unknown bugs in a recent version of the widely used LLVM toolchain, two of which have been confirmed.

Finally, our last contribution is a novel assembly code model named BinBert. This model is built on a transformer pre-trained on a huge dataset of both assembly instruction sequences and execution information (i.e., symbolic expressions). BinBert can be applied to assembly instruction sequences, and it is fine-tunable—that is, it can be retrained as part of a neural architecture on task-specific data. Through fine-tuning, BinBert learns how to apply the general knowledge acquired during pre-training to the specific task. We evaluated BinBert on a multi-task benchmark that we specifically designed to test the understanding of assembly code. The benchmark is composed of several tasks, some taken from the literature and a few novel tasks that we designed, with a mix of intrinsic and downstream tasks. Our results show that BinBert outperforms state-of-the-art models for binary instruction embedding, raising the bar for binary code understanding.

Moreover, BinBert has been developed by taking into account the gaps that we found in our systematization effort—mainly, the lack of comparison with standard architectures, the use of tokenization strategies without comparison and rationale, and the testing of models on a single or very limited tasks.

Contents

Chapter 1

Introduction

The exponential growth of software complexity coupled with the rising of heterogeneous architectures, further complicates the process of manual binary code analysis. Despite its complexity, binary code analysis is extremely valuable, particularly in scenarios where direct access to the source code is unavailable, such as with proprietary software, firmware images, and malware samples. Furthermore, the exponential proliferation of internet-connected devices in households and companies (the IoT[1] revolution), frequently designed with inadequate security measures [65], is inevitably causing a significant increase in the attack surface, giving the chance for malicious actors to hamper the security and privacy of both private and public institutions. Examples are hidden backdoors inserted by malicious producers inside IoT devices or vulnerable firmware where a coding flaw at the source code level could propagate through a multitude of devices across diverse architectures. Those vulnerable devices could be exploited by Advanced Persistent Threat (APT) to obtain long-time persistence or by hackers to create for instance large botnets of infected devices to perform massive Distributed Denial of Service (DDoS) attacks. To mitigate these risks, one approach involves conducting extensive analyses of these devices in an attempt to pinpoint their vulnerabilities or shortcomings. Nevertheless, the manual analysis of executables is a challenging and daunting task, demanding the valuable time and expertise of proficient reverse engineers. To make things worse, reverse engineers, particularly malware analysts, have to deal with the rapid evolution and proliferation of malicious executables, targeting governmental and private organizations at an alarming rate. To tackle these challenges, last few years the scientific community has embarked on studying methods for creating automated binary analysis tools based on Deep Learning (DL) to alleviate the workload of human reverse engineers [161] and significantly enhance software security .

These methods have been inspired by a recent research trend that involves applying Natural Language Processing (NLP) techniques for source code analysis, addressing various tasks. These tasks span from code completion and inspection to debugging, showcasing the adaptability of NLP methods in the realm of software development and analysis. This research line is called “Big code” and consists of leveraging massive open-source codebases (“Big code”) to build statistical models of code capable of producing representations of code using Deep Learning (DL)

[1] Internet of Things

techniques [6]. These representations are called embeddings and refer to a mapping of code snippets in a vector space. The resulting embeddings could be used to build tools capable of solving a variety of tasks thus providing support to code analysis. The rationale for employing NLP techniques on code is grounded in the naturalness hypothesis, which highlights the similarities between software and human language.

> **The naturalness hypothesis.** Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools [6].

Motivated by the proliferation and success of works using NLP techniques for source code, the scientific community has also started investigating their applicability to binary code [38, 107, 175], thus leading to a new line of research.

This thesis further explores this research area by first providing a comprehensive Systematization of Knowledge (SoK) regarding solutions that use deep learning techniques to solve various binary analysis tasks. It then proceeds to apply these techniques to a novel task: the detection of debug information bugs in optimized binaries. To our knowledge, the work introduced in this thesis is the first to explore the use of deep learning techniques for this specific task. Finally, this thesis also advances the research towards the creation of a transformer-based pre-trained model capable of achieving state-of-the-art performance on a wide range of downstream tasks. This work is the first to propose such a model that can be fine-tuned on a variety of tasks.

The subsequent section will highlight all the main contributions provided in this thesis.

1.1 Main Contributions

1.1.1 Literature Review of Binary analysis with Deep Learning

The use of deep learning techniques is revolutionazing lots of fields, including binary analysis. However, as often occurs with disruptive technologies, last few years have witnessed to a proliferation of different solutions without much effort toward systematization. In fact, despite being a fundamental and promising area of research, few efforts have been made to systematically organize the various solutions proposed in the field. Specifically, existing surveys either do not focus specifically on deep learning approaches and do not move beyond 2019 [161], or they concentrate on a single task [105]. In this Chapter, we advance the field by offering a comprehensive review that spans nine years of research, up to 2024. We propose a systematization of 54 research papers and we identify a deep learning pipeline common to all these solutions and provide an in-depth analysis of each of its steps, highlighting key trends across the various approaches as well as gaps that need to be further investigated.

This work will be presented in Chapter 3.

1.1.2 Debugging Debug Information with Neural Networks

The correctness of debug information included in optimized binaries has been the subject of recent attention by the research community. Indeed, it represents a practically important problem, as most of the software running in production is produced by an optimizing compiler. Current solutions rely on invariants, human-defined rules that embed the desired behavior, whose violation may indicate the presence of a bug. Although this approach proved to be effective in discovering several bugs, it is unable to identify bugs that do not trigger invariants. In this work, we investigate the feasibility of using Deep Neural Networks (DNN) to discover incorrect debug information. We trained a set of different models borrowed from the NLP community in an unsupervised way on a large dataset of debug traces and tested their performance on two novel datasets that we propose. Our results are positive and show that DNNs are capable of discovering bugs in both synthetic and real datasets. More interestingly, we performed a *live analysis* of our models by using them as bug detectors in a fuzzing system. We show that they were able to report 12 unknown bugs in the latest version of the widely used LLVM toolchain, 2 of which have been confirmed.

This work will be presented in Chapter 4 and is based on the paper [12] published in the IEEE Access Journal.

1.1.3 BinBert: Binary Code Understanding with a Fine-tunable and Execution-aware Transformer

A recent trend in binary code analysis promotes the use of neural solutions based on instruction embedding models. An instruction embedding model is a neural network that transforms assembly instructions into embedding vectors. If the embedding network is able to processes sequences of assembly instructions transforming them into a sequence of embedding vectors, then the network effectively represents an *assembly code model*.

With this work we present BinBert, a novel assembly code model. BinBert is built on a transformer pre-trained on a huge dataset of both assembly instruction sequences and symbolic execution information. BinBert can be applied to assembly instructions sequences and it is *fine-tunable*, i.e. it can be re-trained as part of a neural architecture on task-specific data. Through fine-tuning, BinBert learns how to apply the general knowledge acquired with pre-training to the specific task. We evaluated BinBert on a multi-task benchmark that we specifically designed to test the understanding of assembly code. The benchmark is composed of several tasks, some taken from the literature, and a few novel tasks that we designed, with a mix of intrinsic and downstream tasks. Our results show that BinBert outperforms state-of-the-art models for binary instruction embedding, raising the bar for binary code understanding.

This work will be presented in Chapter 5 and is based on the paper [13] published in the Transactions on Dependable and Secure Computing (TDSC) journal.

1.2 Thesis Overview

This thesis is structured as it follows:

- Chapter 2 offers an overview of the main deep learning techniques employed in the related literature;
- Chapter 3 provides a structured analysis of the literature of binary analysis with deep learning;
- Chapter 4 presents the paper named Debugging Debug Information with Neural Networks published in the IEEE Access Journal;
- Chapter 5 presents the paper named BinBert: Binary Code Understanding with a Fine-tunable and Execution-aware Transformer published in the Transactions on Dependable and Secure Computing (TDSC) journal.

Chapter 2

Background

2.1 Deep Learning Architectures

Deep Learning is a branch of Artificial Intelligence (AI) that uses machine learning models known as Neural Networks (NNs). These models have a structure that resemble the human brain and are widely used nowadays to solve problems in a variety of fields, ranging from image [52] and audio [171] processing, NLP [172], and program analysis [6].

This Section will introduce the most used models, starting from simpler models like Feed Forward Neural Networks (FFNs) and advancing to more complex and recent architectures like Transformers. The networks that will be presented differ not only in their structure but also in the kind of inputs they are designed for. For instance, Graph Neural Networks (GNNs) are designed to deal with data organized as graphs, Convolutional Neural Networks (CNNs) are more suitable for grid-like data, while Recurrent Neural Networks (RNNs) and Transformers are used for handling sequences.

2.1.1 Feed Forward Neural Networks

FFNNs, also known as Multi-Layer Perceptron (MLP), are one of the simplest neural network. The term "forward" indicates that the input progresses directly to the output without any form of feedback connection. FFNNs aim to learn a function $\hat{f}$ that approximates a target function f by learning parameters θ such that given an input $\mathbf{x}$ the network produce an output $\mathbf{y} = \hat{f}(\mathbf{x}; \theta)$ such that $f(\mathbf{x}) = \mathbf{y}$. As shown in [69], FFNNs are universal approximators and thus are capable of approximating any continuous function. To do so, FFNNs are composed of basic elements known as perceptron, also called neurons, organised in layers (Figure 2.1).

Each neuron acts as a fundamental computational unit within a neural network. It operates by receiving multiple inputs which are multiplied by corresponding weights. After all inputs have been weighted, the neuron sums these values together, also adding a bias term. The obtained sum is then typically passed through an activation function, which determines the neuron's output (Figure 2.2). Activation functions in neural networks are used for two primary purposes. Firstly, they introduce non-linearity, thus giving the network the possibility to go beyond simple linear relationships. Secondly, activation functions determine whether or not a

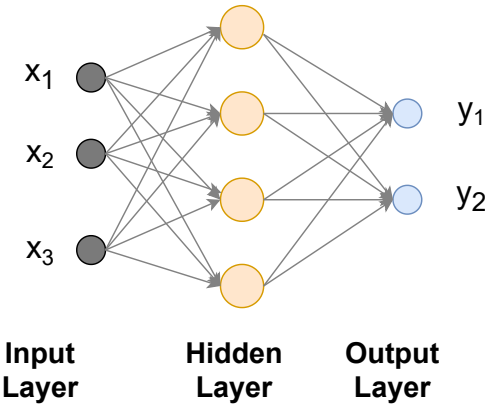


Figure 2.1. Example of FFNN with six neurons (blue and orange dots) organized in two layers: the hidden layer, composed by four orange neurons, and the output layers, composed by two blue neurons.

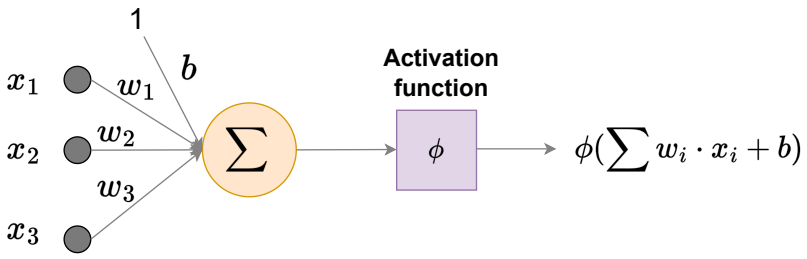


Figure 2.2. Example of a neuron.

neuron should activate in response to a given input. Some of the most widely used activation functions are: binary step, sigmoid, and Rectified Linear Unit (RELU).

Neural networks are typically represented in a matrix/vector form. Thus, given an input vector $\mathbf{x} \in \mathbb{R}^n$ and a weight vector $\mathbf{w} \in \mathbb{R}^n$ and a bias scalar b, a neuron is typically represented in the following form:

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b) \tag{2.1}$$

Similarly, a single-layer FFNN with k neurons can be represented in matricial form as:

$$\mathbf{y} = \phi(W^\intercal \cdot \mathbf{x} + \mathbf{b}) \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input vector, $W \in \mathbb{R}^{n \times k}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^k$ is the bias vector.

Generalizing to FFNNs with l layers, we obtain:

$$\mathbf{y} = \phi_l(W_l^\intercal \cdot \phi_{l-1}(W_{l-1}^\intercal \cdot (\dots \phi_1(W_1^\intercal \cdot \phi_0(W_0^\intercal \cdot \mathbf{x} + \mathbf{b_0}) + \mathbf{b_1}) + \dots) + \mathbf{b_{l-1}}) + \mathbf{b_l}) \tag{2.3}$$

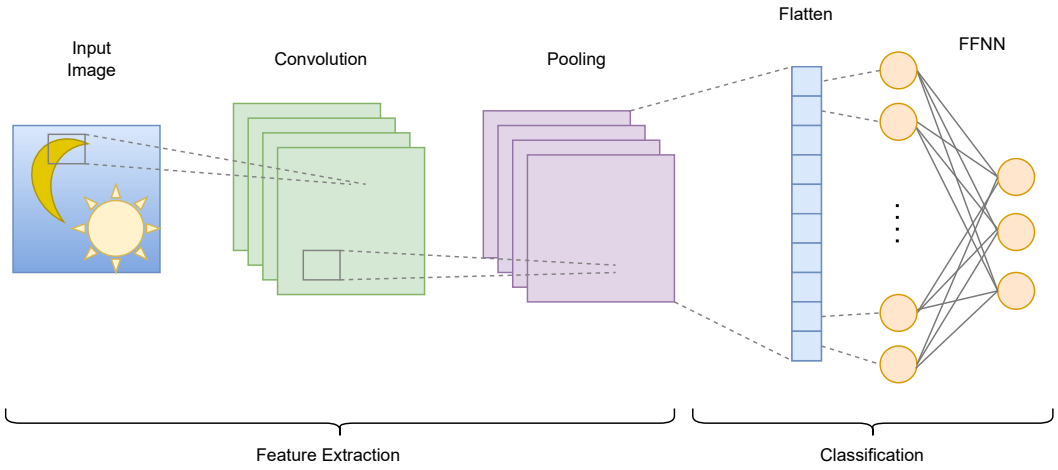


Figure 2.3. Structure of a CNN.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are designed to process grid-like data, for this reason they are widely used in computer vision for image processing applications [52]. The term "convolution" refers to the convolution operation used in mathematics to combine two functions and to describe how one function can affect the shape of another one.

In the context of CNNs, the input data is decomposed into channels, which, in the case of images, correspond to pixels with the Red, Green, and Blue (RGB) components. As shown in Figure 2.3 CNNs are composed of the following layers:

- Convolutional layer: since the input data consists of discrete values (e.g. pixels in an image), discrete convolution is applied. In particular, this layer applies a kernel also known as a filter or feature extractor to the input image. A kernel is a small matrix (often 2D for single-channel inputs or 3D for multi-channel inputs like colored images) that slides over the input image and performs an element-wise multiplication with the corresponding section of the image, followed by summation and an activation function. The goal of the convolution is to extract meaningful features from the input.

- Pooling layer: this layer is used to reduce the dimensionality of the input to decrease the number of parameters required for training and adapt it to the available hardware. Different pooling operations can be applied. Common types of pooling are max and average pooling which compute the max or the average of the image section identified by the kernel.

- Fully Connected layer: in this layer, the output of the pooling layer is flattened and given as input to a regular FFNN for classification purposes.

It is worth highlighting that, multiple convolutional and pooling layers can be used before applying the fully connected layer.

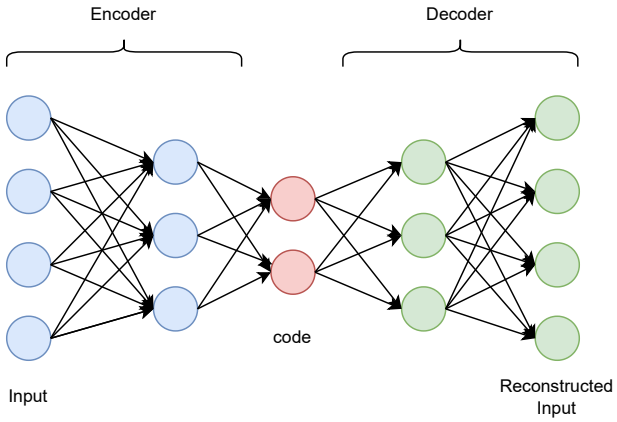


Figure 2.4. Structure of an autoencoder.

2.1.3 Autoencoders

Autoencoders are a type of neural network designed to learn a compressed, latent representation of the input data. In addition to dimensionality reduction, autoencoders are also employed for anomaly detection and data-denoising applications. They are composed of an *encoder* function f that outputs a latent representation $h = f(x)$ known as *code*, while the *decoder* function g produces as output a reconstruction of the input $r = g(h)$ (Figure 2.4), with the aim of having $x \sim r$. Note that autoencoders do not aim at reproducing the input exactly but rather at resembling it. In fact, the reconstruction is typically a close approximation of the original input, highlighting the most significant features. Since autoencoders learn to reconstruct the input from the latent representation without the need for externally provided labels, they are a form of unsupervised learning, and more precisely *self-supervised* due to the use of the input data itself.

2.1.4 Recurrent Neural Networks

Unlike traditional FFNNs, where the input progresses linearly toward the output, RNNs use feedback connections. This design enables them to model sequential data and capture temporal dependencies, as each unit in the network can keep information from previous time steps. For this reason, RNNs have been extensively employed in NLP for processing textual inputs.

RNNs consist of a neural architecture (known as *cell*) that is replicated across each element thus keeping the same parameters throughout the entire sequence. This characteristic, known as *parameter sharing*, ensures that the same computations are applied at every step of the sequence. Each RNN cell receives as input a vector $\mathbf{x}_i$, representing an element of the sequence (e.g. a word in a text), and the hidden state vector at the previous time steps $\mathbf{h}_{i-1}$ and produces an output vector $\mathbf{y}_i$ and the hidden state vector at the current time steps $\mathbf{h}_i$:

$$\mathbf{h}_i, \mathbf{y}_i = RNN(\mathbf{h}_{i-1}, \mathbf{x}_i) \tag{2.4}$$

Hidden states are vectors used to keep information of the previous time steps

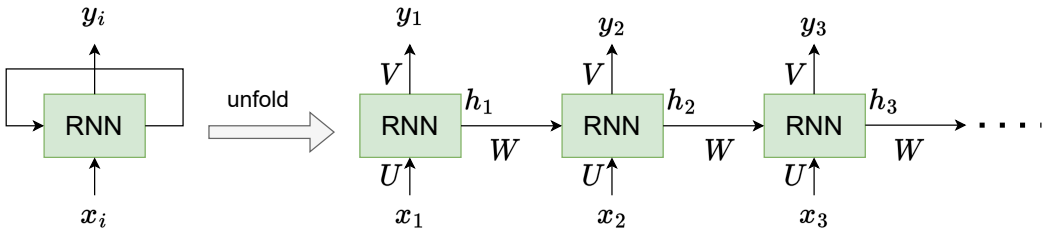
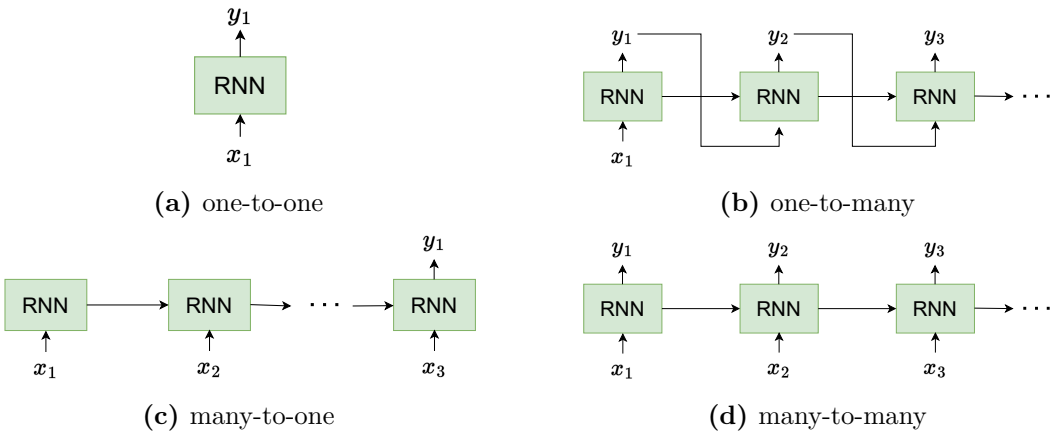


Figure 2.5. Structure of a RNN.



(a) one-to-one **(b)** one-to-many

(c) many-to-one **(d)** many-to-many

Figure 2.6. RNNs design patterns.

and can be considered as the memory of the network. It is worth mentioning that, depending on the specific application, each cell may generate an output vector. However, in scenarios such as sequence classification, the network might be designed to produce a single output only at the end of the sequence.

Due to the concept of *parameter sharing*, a RNN can be visualized in two ways (Figure 2.5):

- In a compact way, known as 'folded' form (shown on the left);
- In an expanded way that reveals the sequence processing, known as 'unfolded' form (shown on the right).

Depending on the underlying applications, RNNs can be arranged according to different design patterns:

- *One-to-one*: produces a single output given a single input (Figure 2.6a);
- *One-to-many*: produces multiple outputs given a single input (Figure 2.6b);
- *Many-to-one*: produces a single ouput given multiple inputs (Figure 2.6c);
- *Many-to-many*: produces multiple outputs given multiple inputs (Figure 2.6d).

The Elman Network

The network shown in Figure 2.5 is known as *Elman Network* and it is one of the simplest type of RNNs. Mathematically speaking an *Elman Network* is defined by the following equations:

$$\begin{aligned} \mathbf{h_i} &= \sigma_h(U \cdot \mathbf{x_i} + W \cdot \mathbf{h_{i-1}} + \mathbf{b}) \\ \mathbf{y_i} &= \sigma_y(V \cdot \mathbf{h_i} + \mathbf{c}) \end{aligned} \tag{2.5}$$

where U, V, W, $\mathbf{b}$ and $\mathbf{c}$ are network parameters and σ_h and σ_y are two non-linear activation functions.

More complex RNNs are Bidirectional Recurrent Neural Networks (BiRNNs) and multilayer RNNs.

Bidirectional Recurrent Neural Networks

As the name suggests, BiRNNs are structured to process data in both forward and backward directions. For this reason, this architecture is particularly effective in scenarios where the output at a specific time-step is influenced not only by preceding elements in the sequence but also by subsequent ones. A BiRNN consists of two distinct RNNs:

- One travelling forward $\overrightarrow{RNN}(\cdot)$ that processes the sequence from the start to the end;
- One travelling backward $\overleftarrow{RNN}(\cdot)$ that processes the sequence from the end to the start.

The output of a BiRNN at any time-step is a combination of the outputs from both the forward network ($\overrightarrow{\mathbf{y}}_i$) and the backward network ($\overleftarrow{\mathbf{y}}_i$), typically concatenated together. Mathematically speaking, a BiRNN is described by the following set of equations:

$$\begin{aligned} \overrightarrow{\mathbf{h}}_i, \overrightarrow{\mathbf{y}}_i &= \overrightarrow{RNN}(\overrightarrow{\mathbf{h}}_{i-1}, \mathbf{x}_i) \\ \overleftarrow{\mathbf{h}}_i, \overleftarrow{\mathbf{y}}_i &= \overleftarrow{RNN}(\overleftarrow{\mathbf{h}}_{i-1}, \mathbf{x}_i) \\ \mathbf{y}_i &= [\overrightarrow{\mathbf{y}}_i; \overleftarrow{\mathbf{y}}_i] \end{aligned} \tag{2.6}$$

Multi-layer Recurrent Neural Networks

Figure 2.8 illustrates how complexity can be added to a neural network architecture to enhance performance by stacking multiple RNNs on top of each other. This layered configuration results in what is known as a Multi-layer RNN.

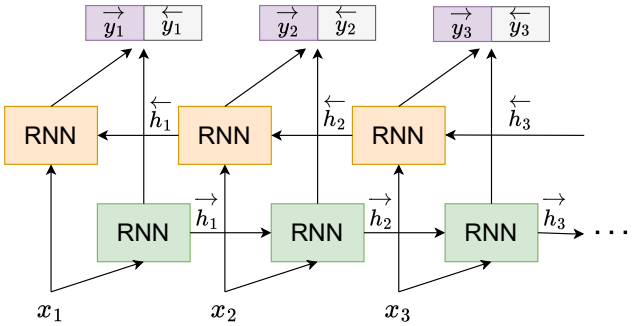


Figure 2.7. Structure of a BiRNN.

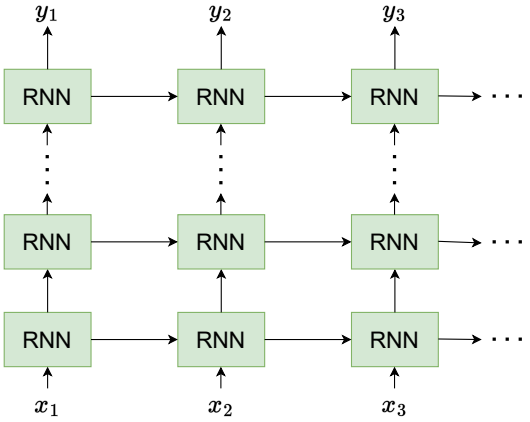


Figure 2.8. Structure of a multi-layer RNN.

The Vanishing Gradient Problem and Gated Architecture

As traditional Deep Learning architectures, RNNs are trained using the backpropagation algorithm which is based on the gradient computation to update network's weights. These gradients, computed with the chain rule, are progressively propagated backward through the network's layers. For long sequences, gradients associated with early stages can become exponentially smaller. This can lead the network to forget early inputs, thus making it unable to handle long-term dependencies properly. This problem is known as the *vanishing gradient problem*.

To address such problem, Long-Short-Term-Memory (LSTM) and Gated Recurrent Unit (GRU) have been introduced. LSTMs consist of some gates used to regulate which part of the input information has to be read (input gate), which information coming from previous time steps has to be forgotten (forget gate) and which information has to be sent as output (output gate). This selective memory process helps LSTMs to preserve long-term dependencies and mitigate the vanishing gradient problem. GRUs have similar structures to LSTM but they have less parameters and thus they allow faster training.

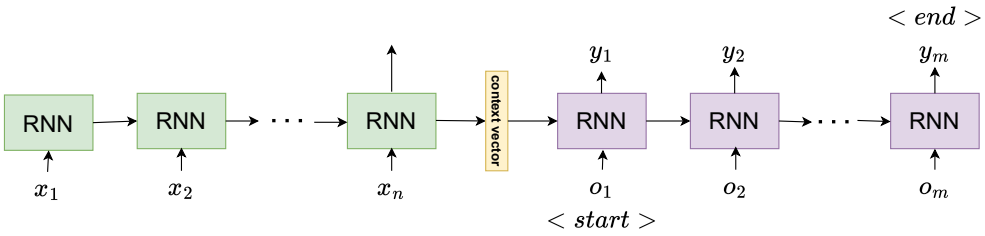


Figure 2.9. Sequence-to-Sequence network.

2.1.5 Sequence-to-Sequence Architectures

As depicted in Figure 2.6, traditional RNNs can map a sequence of inputs to a sequence of outputs of the same lenghts. However, some specific applications require an output sequence which have a different size with respect to the input one. To address this limitation, Sequence-to-Sequence (seq2seq) networks have been introduced. These networks, also known as Encoder-Decoder networks, have been designed to handle situations where the input and output sequences are not aligned in length, making them suitable for tasks like machine translation, where the length of the translated output may not correspond the input one.

The seq2seq framework, first introduced by [139, 140] and shown in Figure 2.9, consists of two main components:

- *Encoder*: based on a RNN, it processes an input sequence of size n and condenses the information into a fixed-size *context vector* $\mathbf{c}$, capturing the key elements of the input. The *context vector* $\mathbf{c}$ can be the hidden layer of the last cell or a non-linear combination of the hidden states of the whole sequence;
- *Decoder*: based on a RNN, it subsequently utilizes the context vector to generate the output sequence of size m. Specifically, the first RNN cell takes as input an element denoting the start of the sequence, while the last cell produces as output another element determinig the end of the sequence ($< start >$ and $< end >$ in Figure 2.9). It is important to note that during inference, the output produced by the sequence at a given time step is used as the input for the subsequent time step. However, during training, to prevent the propagation of errors through the time steps, the true input o_i is provided at each step instead. This training strategy is referred to as *teacher forcing*.

The two RNNs are simultaneously trained to maximize the probability of the output sequence:

$$P(\mathbf{y_1}, \ldots, \mathbf{y_m}) = \prod_{i=1}^{n} p(\mathbf{y_i}|\mathbf{y_1}, \ldots, \mathbf{y_{i-1}}, \mathbf{c}) \tag{2.7}$$

The Attention Mechanism

Because this network depends on a single fixed-size *context vector* to encapsulate the entirety of the input sequence's information, it faces challenges in effectively processing long input sequences. The reliance on such a *context vector* can result

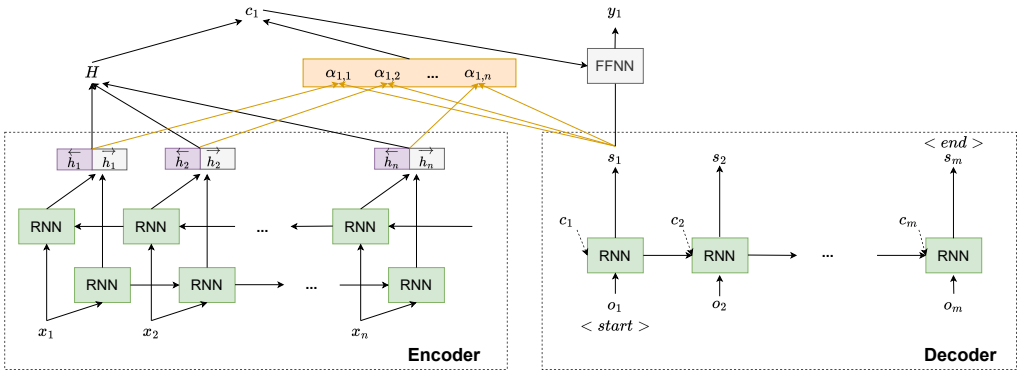


Figure 2.10. Sequence-to-Sequence network with attention.

in the loss of information, as it must compress all input details into a constrained representation.

For this reason, [15] introduced a framework that incorporates the so-called *attention mechanism*. Originally developed within the NLP community to tackle machine translation problem, this approach deviates from the traditional method of compressing input information into a single vector. Instead, the network maintains a series of vectors that represent the relevance of each input element at every step of the decoding process, allowing for a more effective handling of sequence data. As depicted in Figure 2.10, the architecture proposed by [15] consists of a bidirectional encoder (see Equation 2.8) and a unidirectional attention decoder.

The attention decoder is represented by the following set of equations:

$$\begin{aligned}
&\mathbf{s_i} = \sigma_h(U \cdot [\mathbf{o_i}; \mathbf{c_i}] + W \cdot s_{i-1} + \mathbf{b_s}) \\
&\mathbf{c_i} = H \cdot \alpha_\mathbf{i}, \text{ where } H = [\mathbf{h_1}; \mathbf{h_2}; \ldots; \mathbf{h_n}] \\
&\alpha_\mathbf{i} = (\alpha_{i,1}, \alpha_{i,2}, \cdots, \alpha_{i,n}), \text{ where } \alpha_{i,j} = softmax(attn_score(\mathbf{s_{i-1}}, \mathbf{h_j})) \\
&\mathbf{y_i} = \sigma_y(V \cdot [\mathbf{s_i}; \mathbf{c_i}] + \mathbf{b_y})
\end{aligned} \tag{2.8}$$

To generate the hidden state representation $\mathbf{s}_i$, the attention decoder leverages the hidden state from the previous time step $\mathbf{s}_{i-1}$, the representation of the preceding element $\mathbf{o}_i$, and the *context vector* $\mathbf{c}_i$. This *context vector* is derived from a weighted sum of the encoder's hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_n$, where the weights are determined by applying the softmax function to the attention scores. These scores, calculated by an attention scoring function, assess the relevance of the input at position j to the output at position i. The scoring function itself is a feedforward neural network that is trained in conjunction with the entire network.

2.1.6 Transformers

In 2017, [149] introduced a novel architecture for processing sequential data known as Transformer. As for the the attention-based seq2seq network, the Transformer was originally developed within the NLP community to solve the machine translation problem. However, the Transformer architecture rapidly became a new standard for a variety of NLP tasks beyond machine translation, including text summarization,

question-answering, and language understanding, achiving state-of-the-art results across a plethora of NLP benchmarks.

Unlike its predecessors that relied on recurrence, the Transformer model is entirely based on a mechanism called self-attention, which allows it to directly compute relationships between all parts of the input sequence, regardless of their positions. Since it does not rely on recurrence, this architecture facilitates parallel processing of data, significantly enhancing efficiency and scalability compared to RNNs and LSTMs. Additionally, the Transformer mitigates the vanishing gradient problem, enabling it to better handle long sequences.

As shown in Figure 2.11, the transformer network consists of a set of N stacked encoders and a set of N stacked decoders. In particular, the encoder stack maps the input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$ to vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$, used by the decoder stack to generate the output sequence, one element at a time.

Encoder Stack

The encoder is composed of a stack of N identical layers. The bottom-most layer is fed with input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$ whereas all the other layers are fed with the output of the encoder directly below. Each layer consists of two sub-layers:

- A *multi-head self-attention* mechanism used to understand which part of the input matters while processing a specific element of the sequence;
- Fully connected feed-forward networks independently applied to each position.

Moreover, Each sub-layer has a residual connection around it, followed by a layer-normalization step. This design facilitates the flow of gradients during training, enhancing the model's learning efficiency. Notably, the Transformer diverges from traditional seq2seq models since each token flows through a distinct path within the encoder: while dependencies among tokens exist in the self-attention layers, the feed-forward layers do not have these dependencies. This specific design enable parallel processing.

Decoder Stack

The decoder is structured as a stack of N identical layers. The bottom-most layer receives as input the output tokens shifted by one position, ensuring that at each time step the next token in the sequence is predicted. Subsequent layers are fed with the output from the layers directly beneath them. Each layer consists of three sub-layers:

- A *masked multi-head self-attention* mechanism, which operates similarly to the self-attention mechanism of the encoder, with the addition of masking. Masking is crucial during the decoding process as it prevents the mechanism from attending to subsequent tokens, ensuring that the prediction for a given position can only be influenced by tokens already seen by the network;
- A *multi-head attention* mechanism which performs attention with the output of the top-most layer of the encoder. This mechanism enables the decoder to selectively prioritize information from different parts of the encoder's output;

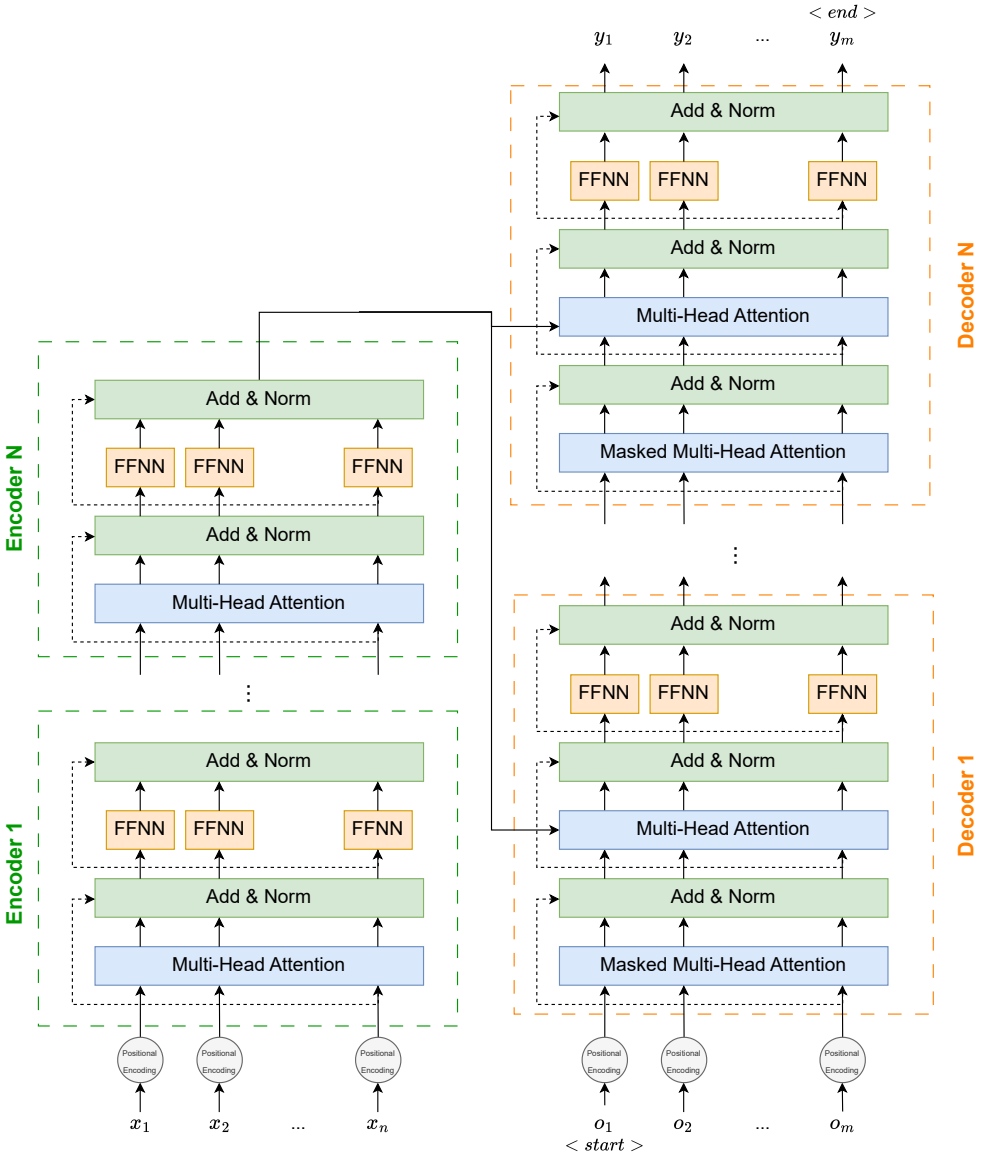


Figure 2.11. Transformer architecture.

- Fully connected feed forward networks.

Similar to the encoder, a residual connection around each sub-layer is used, followed by a layer normalization.

Attention

The Transformer architecture is characterized by three distinct types of attention mechanisms: multi-head self-attention within the encoder, masked multi-head self-attention within the decoder, and multi-head attention (also known as encoder-decoder attention), situated in the decoder, which mirrors the traditional attention mechanism of the seq2seq models. To fully understand these attention mechanisms, it is essential to first grasp the foundational concepts of query, key, and value, which form the basis of how attention is computed within the Transformer model. As already explained above, in the attention-based seq2seq model, the context vector $\mathbf{c}_i$ is derived through a weighted linear combination of encoder hidden states, utilizing attention scores α_i as weights. This context vector, in conjunction with the hidden states, is then leveraged to generate the output sequence. In the Transformer network, the attention mechanism is slightly different. Each input element x_i is represented by a pair comprising a key and a value vector, each with dimensions d_k and d_v, respectively. Meanwhile, each output token is conceptualized as a query vector of dimension d_k. The determination of attention scores involves calculating the similarity between each query and the corresponding keys. Subsequently, the context vector is generated through a process similar to earlier models: by forming a weighted linear combination of the value vectors, using the attention scores as the weighting coefficients. Specifically, the Transformer employs a form of attention known as *Scaled Dot-Product Attention*, which is calculated using the formula:

$$Attention(\boldsymbol{q}_i, K, V) = \sum_{s=1}^{n} \frac{1}{z} exp(\frac{< \boldsymbol{q}_i, K_{s,\cdot} >}{\sqrt{d_k}}) V_{s,\cdot} \tag{2.9}$$

In this equation, $K \in \mathbb{R}^{n \times d_k}$ and $V \in \mathbb{R}^{n \times d_v}$ represent the matrices containing the key and value vectors of the input elements in their rows, respectively. When the query vectors are aggregated into a single matrix $Q \in \mathbb{R}^{m \times d_k}$, it is possible to rewrite the equation above as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.10}$$

The division $\sqrt{d_k}$ is necessary to prevent inner product from becoming too large.

The attention mechanism is enhanced through the introduction of a *multi-headed attention*. This improvement allows the mechanism to simultaneously focus on different parts of the input and understand it from various perspectives or subspaces. Instead of using a single set of keys, values, and queries, multi-headed attention creates multiple sets (or heads) of these elements, each projecting the queries, keys, and values into different dimensions. This means that the attention process is carried out several times in parallel, each time looking at the information in a slightly different way. The results of these parallel attention processes are then combined

and once again projected to form the final output. This method allows the model to capture a richer array of information from the input:

$$\begin{aligned} MultiHead(Q,K,V) &= [head_1;\ldots,head_h]W^O \\ \text{where } head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.11}$$

W_i^Q, W_i^K, and W_i^V are matrices that transform the input into different dimensions for each head, and W^O is a matrix that combines the outputs from all heads into the final output.

As previously mentioned, the transformer uses three variants of attention:

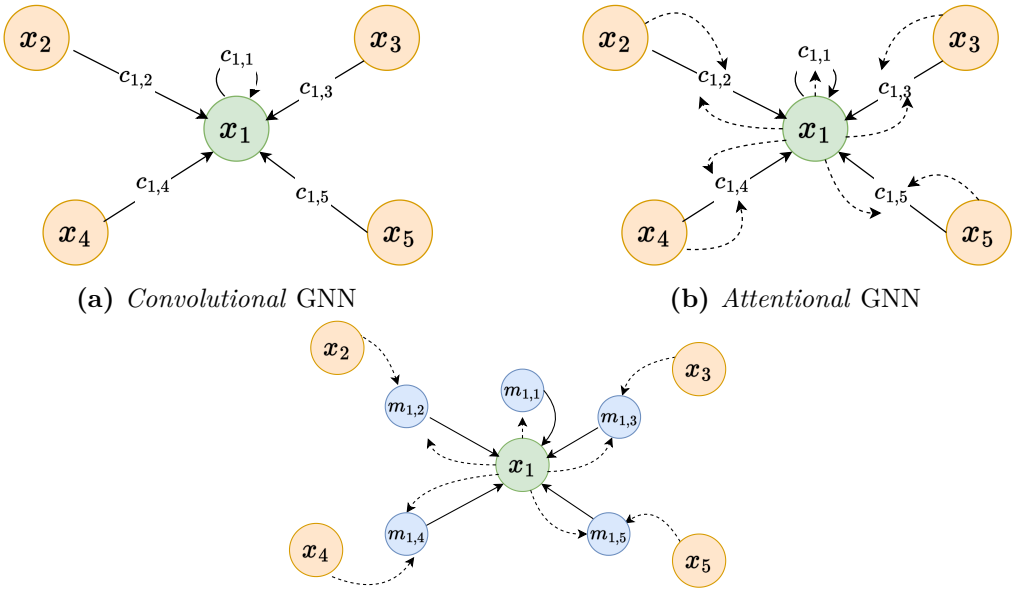
- *Encoder Multi-Head Self-Attention*: In this case, queries keys and values are taken from the output of the previous layer in the encoder. Each position in the encoder attends to all the other positions in the previous layer of the encoder;
- *Decoder Masked Multi-Head Self-Attention*: This attention is similar to the one described above in the sense that queries, keys and values come from the same place, i.e. from the output of the previous decoder. Each position in the decoder attends to all the other positions in the previous layer of the decoder, except for subsequent positions. This is achieved through masking;
- *Encoder-Decoder Attention*: This attention is similar to the attention mechanism of traditional seq2seq network. Queries come from the output of the previous decoder layer, whereas keys and values come from the output of the bottom-most encoder. This setup allows each position within the decoder to consider all positions of the input sequence.

Position-wise Feed-Forward Networks

Within both the encoder and decoder, each layer includes a feed-forward network that operates on each position independently. The linear transformations are consistent across all positions, whereas the parameters of these transformations vary from one layer to another.

Positional Encoding

Due to the absence of recurrence, which intrinsically accounts for the sequential order of inputs, the Transformer architecture, by itself, lacks an effective mechanism able to recognize this order. In fact, it leverages multi-head attention and position-wise feed-forward networks to independently compute the output for each position within the sequence. While this design enables parallel computation, it does not directly model the sequential order. To address this, the Transformer incorporates positional information into the input embeddings at the initial layer of both the encoder and decoder. Specifically, it utilizes positional encoding to generate real-valued vectors for each input element, encapsulating the order information. These vectors are then added to the corresponding input vectors, so as to make the model aware of the sequential order.



(a) *Convolutional* GNN **(b)** *Attentional* GNN

(c) *Message Passing* GNN

Figure 2.12. GNNs

2.1.7 Graph Neural Networks

In recent years, literature has highlighted a growing number of application domains where input data is inherently graph-based, such as social network analysis [54, 63], biology [59], and more. This has prompted researchers to leverage the capabilities of neural networks for the representation of graph-structured data effectively.

Drawing inspiration from the progress in word embeddings within the NLP field, numerous studies have emerged to tackle the challenge of graph representation learning (also known as graph embedding). In fact, after the introduction of *word2vec* [144] used to represent words in the vector space (see Section 2.2.1), several works started employing similar techniques to graphs. For instance, the *DeepWalk* model [124] treat nodes in a graph similarly to words in a natural language by applying techniques from the word embedding domain. Specifically, *DeepWalk* first generate random walks in a graphs and then applies the *word2vec* model to learn representations for each node. To do so, the model is designed to predict the neighboring nodes in a walk given the current node. However, besides structural information, each node could contain rich textual information that should be taken into account to create node emebeddings. To tackle this challenge [162] introduced an extension of the original *DeepWalk* method known as *Text-associated DeepWalk* (TADW) algorithm. This enhancement allows the creation of more complex embeddings capable of reflecting both the network topology and the semantic content of each node.

After the advent of random walk-based methods for graph representation, various Graph Neural Networks (GNNs) have been introduced to significantly advance the field by providing a more direct and powerful approach to learning from graph-structured data [150, 173]. GNNs operate through an iterative process that continu-

ously augment the states of nodes based on the information from their neighbors. Initially, the process incorporates data from immediate neighbors, then from nodes two hops away, progressively collecting the information with each subsequent iteration. This process ensures that the representation of each node is shaped not only by its own attributes but also by those of its adjacent nodes. Moreover, at each iteration neural networks are applied to each aggregated information to finalize the node representation. Final representations are then leveraged to address specific problems, including node and graph classification. GNNs can be cathegorized as follows [150]:

- *Convolutional* GNN: these networks are designed to extend the convolution operation to graph-structured data [83]. In such networks, features from neighboring nodes N_i are aggregated, employing fixed, precomputed weights $c_{i,j}$ (see Figure 2.12a). *convolutional* GNNs can be represented by the following mathematical formulation:

$$x_i^{t+1} = \sigma_h(x_i^t, \bigoplus_{j \in N_i} c_{i,j} \cdot \sigma_n(x_j^t)) \tag{2.12}$$

  where x_i^t is the node representation at time step t and σ_h and σ_n are two Neural Networks;

- *Attentional* GNN: these networks incorporate the attention mechanism, originally introduced in the NLP domain by [15], to enhance graph-structured data processing. Mimicking the way the attention mechanism assigns weights to words based on their importance within a sentence in NLP, *attentional* GNNs apply this concept to graph nodes [151]. This approach enables the model to dynamically assess and prioritize the importance of each neighboring node during the feature aggregation process. *attentional* GNNs can be represented by the following mathematical formulation:

$$x_i^{t+1} = \sigma_h(x_i^t, \bigoplus_{j \in N_i} attn\_score(x_i^t, x_j^t) \cdot \sigma_n(x_j^t)) \tag{2.13}$$

- *Message Passing* GNN: Unlike the previously mentioned networks, which rely on features weighted by some constants, this network adopts a collaborative approach by computing messages in conjunction with neighboring nodes [58]. These messages are subsequently utilized to update the features of the nodes. These kind of networks implement a more flexible approach to learn graph representation and can be represented by the following mathematical formulation:

$$x_i^{t+1} = \sigma_h(x_i^t, \bigoplus_{j \in N_i} \sigma_n(x_i^t, x_j^t)) \tag{2.14}$$

  where $\sigma_n(x_i^t, x_j^t)$ is the message created and exchanged between node i and node j.

2.2 Language Models

Language modeling is the primary approach used in NLP to capture language understanding, enabling machines to mimic and reproduce human language comprehension. Technically speaking, a language model is a statistical distribution of words in a sequence that facilitates the prediction of the next word in that sequence or of a word in a specific context. Given a sequence of $n-1$ words belonging to a vocabulary V, a language model aims at computing the probability of the n^{th} word:

$$P(w_n|w_1,\ldots,w_{n-1}) = \frac{P(w_1,\ldots,w_{n-1},w_n)}{P(w_1,\ldots,w_{n-1})} \tag{2.15}$$

The probability $P(w_1,\ldots,w_n)$ can be obtained using the chain rule:

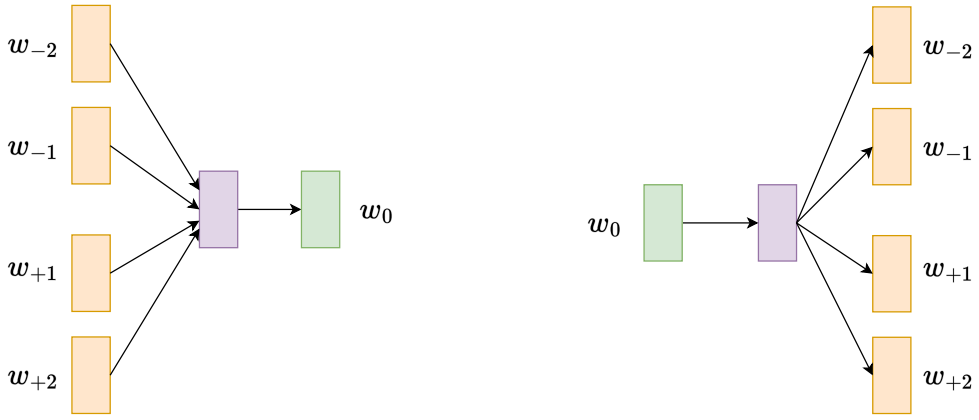
$$\begin{aligned} P(w_1,\ldots,w_n) &= \prod_{i=1}^{n} P(w_i|w_1,\ldots,w_{i-1}) = \\ &= P(w_1)\cdot P(w_2|w_1)\cdot P(w_3|w_2,w_1)\ldots P(w_n|w_1,\ldots,w_{n-1}) \end{aligned} \tag{2.16}$$

As the context expands, modeling these probabilities becomes increasingly challenging, necessitating simplifying assumptions. One such simplification is the Markovian assumption, according to which a given word w is influenced solely by its preceding word. Consequently, the probability $P(w_1,\ldots,w_n)$ transforms into the product of bigram probabilities. This assumption can be further generalized to assume that a word w depends on the previous n words, thus leading to the product of n-gram probabilities. These approaches are known in the literature as *Statistical Language Modeling* sinche n-gram probabilities are estimated from a corpus of documents. However, an inherent limitation of such approaches arises from the fact that not all sequences, even if legitimate, appear in the training corpus, causing some probabilities to be zero. This problem is known in the literature as the *curse of dimensionality*.

2.2.1 Neural Language Models

Given the inherent limitations of *Statistical Language Modeling* approaches, [23] introduced a solution to address the curse of dimensionality problem by employing a Feed-Forward Neural Network. Moreover, [23] pioneered the idea of *distributed word representations*, wherein each vocabulary word is associated with a vector in the d-dimensional space. These vectors, referred to as *word embeddings*, are designed to encapsulate semantic information about the respective words. The computation of the probability distribution for the next word in a sentence involves feeding the neural network with a combination of the embeddings of all the words in the sentence.

Later, [144] introduced a more efficient Neural Network, referred to as *word2vec*, for computing word embeddings. *Word2vec* is a 2-layer Feed Forward Neural Network specifically designed to produce embeddings in a manner that aligns with the semantic similarity of words: words with similar meanings are assigned vectors that are close in the vector space. *Word2vec* is trained in an unsupervised way and depending on



(a) C-BOW **(b)** Skip-Gram

Figure 2.13. *Word2vec* training approaches.

the specific tasks it is trained on, it is possible to distinguish two configuration (see Figure 2.13):

- C-BOW: the network is fed with context words (m preceding and m subsequent words) as input and it has to produce the target word as output. In this case, given a dataset of T words, the goal is to minimize the following loss function:

$$\frac{1}{T}\sum_{i=1}^{T}\sum_{-m\leq j\leq m, i\neq 0}\log P(w_i|w_{i+j}) \tag{2.17}$$

- Skip-Gram: the network is fed with a target word as input and it has to predict its context words as output (m preceding and m subsequent words). In this case, given a dataset of T words, the goal is to minimize the following loss function:

$$\frac{1}{T}\sum_{i=1}^{T}\sum_{-m\leq j\leq m, i\neq 0}\log P(w_{i+j}|w_i) \tag{2.18}$$

2.2.2 Pre-trained Language Models

A fundamental constraint of word embedding models, such as *word2vec*, lies in their inability to effectively model polysemous words. This limitation arises because a single vector is employed to represent a word, regardless of its various meanings in different contexts. Moreover, since words typically have predominant meanings, their vector representations are biased toward their most common sense.

To address these limitations, [125] introduced a model named *ELMo*, designed to generate context-sensitive word embeddings. This innovative model comprises a 2-layer bidirectional LSTM network, pre-trained with the objective of predicting the next word in a given sentence. Notably, *ELMo* played a pivotal role in introducing the concept of the pre-training and fine-tuning paradigm to NLP. This paradigm

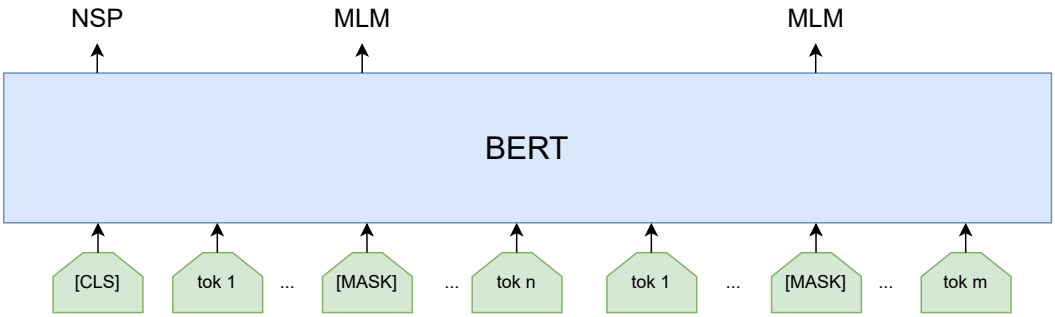


Figure 2.14. Bert model.

consists in initially pre-training a model on an unsupervised task, enabling it to learn contextualized representations, and subsequently fine-tuning the model on a specific downstream task. This two-step process has proven effective in leveraging learned contextual information to enhance the overall performance of various natural language understanding applications.

BERT

To enhance the representation of word embeddings in context, [44]developed a model called BERT (Bidirectional Encoder Representations from Transformers). BERT processes a sentence to produce contextualized vector representations for each word. Its architecture, built on the Transformer encoder, is pre-trained through two unsupervised tasks:

- Masked Language Modeling (MLM): This task involves masking a certain percentage of the input sentence's tokens and making the model to predict the masked ones. Notably, the model utilizes the context provided by both preceding and subsequent words to accurately predict the masked token;
- Next Sentence Prediction (NSP): This task aims to determine whether two sentences are sequentially related. During training, for half of the input pairs, the second sentence is the true following sentence, while for the other half, it is randomly selected from the corpus.

After pre-training, the BERT model can be fine-tuned on specific downstream tasks.

BERT utilizes the WordPiece tokenization method, which begins with a basic vocabulary that includes special tokens and the initial alphabet. This vocabulary is incrementally merged to achieve the desired size. The special tokens are:

- CLS: Added at the beginning of each input sequence, its final hidden state can serve as an aggregate representation for the entire sequence;
- SEP: Used to delineate two input sentences in the NSP task;
- MASK: Employed to obscure tokens in the sentence for the MLM task.

2.3 Binary analysis and Reverse Engineering

Binary analysis is the process of analysing binary code to pinpoint its functionality and behaviour, particularly when the source code is inaccessible. This critical process is applied in various scenarios, including identifying bugs or vulnerabilities in proprietary software, applying patches to software in production environments, or comprehensively analysing the functionalities of malicious executables. To achieve this, a range of binary analysis techniques have been developed, which can be broadly classified into two categories: static and dynamic analysis. Static analysis techniques do not require running the binary, thus offering more flexibility since they can be analyzed on any CPU architecture. On the other hand, dynamic analysis requires running a binary in a specific isolated environment running on an architecture capable of supporting, or emulating, that binary format. Although this method is less versatile, it simplifies the analysis process by enabling real-time observation of the binary's behavior.

It is important to note that achieving a comprehensive understanding of a binary's characteristics often involves manual code reversing. Despite being one of the more challenging aspects of binary analysis, it remains the most effective method for uncovering the program's internal logic. This process typically requires the use of a disassembler to retrieve assembly instructions for analysis, potentially complemented by a decompiler to gain a higher-level code perspective, and a debugger to monitor the program's behavior during execution.

Binary analysis and reverse engineering are notably challenging tasks. A significant factor contributing to their complexity is the common practice of *stripping* binary code to protect proprietary software from being copied or to make the analysis of malware more difficult. Stripping involves removing all high-level information from the symbol table, such as variable names and types, function names, and the number and type of function arguments. This information is helpful for analysts to fully understand the code's functionality. Additionally, the same source code compiled across different architectures and optimization levels can result in vastly different binary outputs, further complicating the analysis. Compounding these challenges, binaries frequently undergo obfuscation, deliberately increasing the difficulty of binary analysis.

2.3.1 The Compilation Process

Figure 2.15 delineates the sequential stages of the compilation toolchain used to transform high-level source code into machine code, which is subsequently interpreted and executed by the CPU. The initial stage involves compilation, where the source code is transformed into object files. These object files contain machine code that is not bound to any specific memory address, rendering them incapable of direct execution. Importantly, object files may include unresolved references to symbols found in other object files or libraries. To address these unresolved references, the linker comes into play. It takes all the object files associated with a program and put them into a singular executable. This executable can be loaded at a specific memory address for execution. Additionally, the linker plays a crucial role in resolving symbolic references present in other object files or statically linked libraries. Unlike

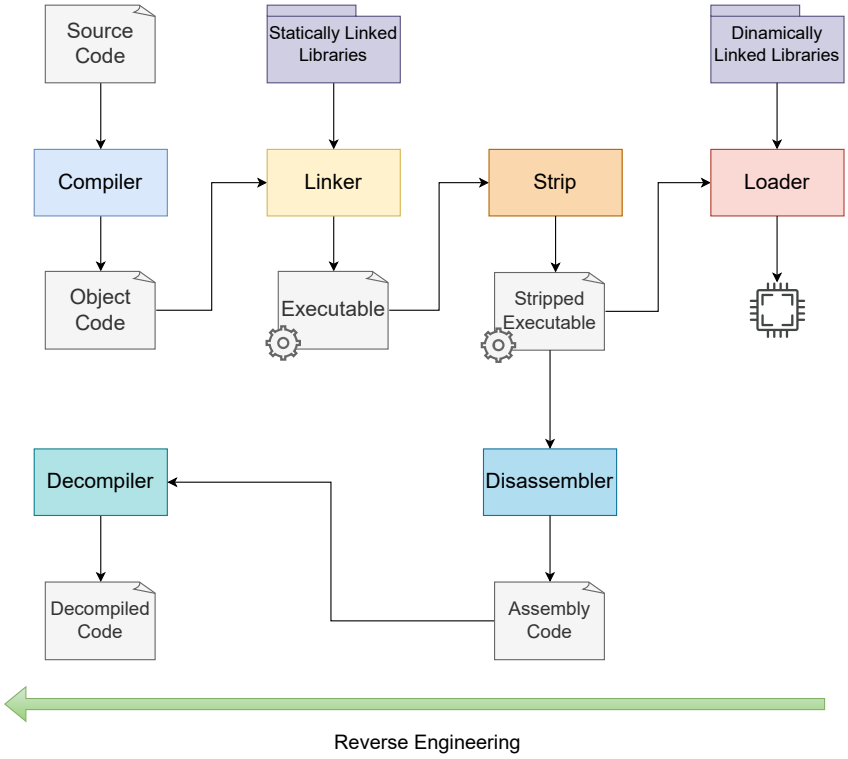


Figure 2.15. The compilation toolchain and reverse engineering.

dynamically linked libraries, which are loaded at runtime, statically linked libraries are integrated into the binary itself.

2.3.2 Dissassemblers

Disassembly is the process of converting a bytecode into an higher level representation consisting of assembly instructions. The disassembly process is usually performed with a mixture of algorithimic techniques and heuristics aimed at differentiating the actual code from the embedded data. As a matter of fact, binaries often include inline data, which may lead the disassembler to misinterpret them as code, resulting in the generation of invalid instructions or, in extreme cases, causing the entire disassembly process to become desynchronized.

Disassembly techniques can generally be categorized as linear sweep and recursive disassembly. Simpler disassemblers, such as *objdump*, employ linear sweep that decodes bytes consecutively. However, these methods have a drawback, they are prone to misinterpreting data as code, leading to incorrect assembly instructions. On the contrary, recursive disassembly follows the flow of control, mitigating the issue of misinterpreting inline data. Nonetheless, certain portions of the code are challenging to reach statically, such as the code targeted by indirect calls or jumps whose location is revealed only at runtime. Popular tools like IDA, Ghidra, Radare2, Angr, etc., leverage recursive disassembly as it stands out as the most precise technique in the field.

In this regards, the studies conducted by [75, 114, 115] assess the correctness of

commercial disassembly tools, demonstrating that, in the majority of cases, they exhibit good performances. Specifically, they measure the precision and recall of recovering assembly instructions, identifying the function entry points and reconstructing the Control Flow Graph (CFG). According to the experiments of [114] for x86 and x64 binaries, Ghidra, IDA and Radare2 shows a precision ranging between 98% and 99% and a good recall in recovering valid assembly instructions; a precision ranging between 97% and 99% and a recall above 60% in extracting correct function entry points. Regarding CFG reconstruction, while Ghidra and Radare mantains good performances, IDA shows best precision and recall.

2.3.3 Decompilers

Decompilation is the process of reconstructing high-level source code, typically starting from disassembled code. Although decompiled code may contain errors and may not precisely replicate the original source code, it aids human reversers in comprehending the code more easily due to its improved readability. In particular, decompiled code aims to reconstruct high-level constructs such as function signatures, variables, as well as for and if statements. Popular decompiler tools are IDA Hex Rays and the Ghidra decompiler.

2.3.4 Symbolic Execution

Besides static and dynamic analysis techniques, another means for binary analysis is represented by symbolic execution [17]. Instead of executing a program with concrete inputs, symbolic execution treats input values as symbols and computes logical formulas (known as symbolic expression) over these symbols as the program execution proceed. Moreover, to keep track of path constraints imposed by the different execution branches, symbolic constraints are associated to symbolic expressions.

Symbolic execution is particularly helpful for checking the presence of a specific property of a program without directly executing it. Unfortunately it also has some drawbacks. One of the main limitations is the path explosion problem, wherein exploring and managing all potential execution paths becomes impractical due to the exponential increase of paths with the number of branches. Another limitation is represented by the difficulty of modeling the effect of calling external dependencies. While certain approaches attempt to capture the effects of system dependencies, achieving precision in this regard remains a challenging problem.

Chapter 3

Literature Review of Binary analysis with Deep Learning

One of the most exciting trends in the wide research landscape has been the renaissance of DNNs, which have been widely applied across various settings to solve a plethora of tasks ranging from computer vision to natural language processing. The field of binary analysis has not been immune to this compelling trend, adopting novel neural architectures and paradigms such as graph neural networks, transformer architectures, and word embeddings, among others [161]. However, as often occurs when a field is rapidly influenced by the disruptive presence of powerful technology, there has been a swift proliferation of novel solutions to solve disparate binary analysis tasks without much effort towards systematization. Consequently, the current landscape of DNNs applied to binary analysis tasks is fragmented and scattered. This create several problematic aspects that in the long term will hinder the collective research effort, as example: many solutions address the same tasks using partially overlapping ideas without adequate comparison; methodologies and steps that can be applied to several different tasks (to name a few the preprocessing of assembly instructions and their representations) are each time reinvented from scratch; substantially equivalent tasks are defined in slightly different ways making solutions hard or impossible to compare; the datasets on which the models are trained and tested overrepresent some architectures and operating systems totally neglecting others. This effect is exacerbated by the fact that the binary analysis domain spans several communities, including security research, software engineering, and machine learning.

In this chapter, we take an initial step towards systematizing the landscape of deep neural network (DNN) solutions applied to binary analysis tasks. We conducted a survey of current solutions across various communities and defined a pipeline common to all deep learning-based approaches. Building upon this pipeline, we systematize its main building blocks, identifying unresolved gaps in the literature for each. More specifically, we provide a systematization of:

- **Downstream Tasks**: We identified 25 different tasks, along with minor variants. Our analysis reveals that some tasks have been investigated more extensively than others and are sometimes redefined in slightly different ways, complicating comparisons;

- **Raw Datasets**: We analyzed the datasets used in the literature and observed that they over-represent Linux binaries, with only a small representation of Windows executables. Additionally, there is a common trend to recreate datasets from scratch without considering previous work on the same task, making comparisons difficult;

- **Binary Representation**: We identified all the different binary representation methods and discovered that, besides traditional representations (e.g., Control Flow Graphs), many exotic representations have been proposed. Unfortunately, comparisons between these custom representations and standard ones are missing. Therefore, further research is needed to pinpoint the most suitable representations;

- **Preprocessing and Tokenization**: All the analyzed solutions define preprocessing and tokenization rules capable of reducing the vocabulary size. A common trend for tokenization is to split on whitespace, punctuation, or assembly instructions. However, automatic tokenization methods, such as Byte Pair Encoding (BPE), are becoming the standard in this field since they show better performance [13, 82]. Regarding preprocessing, most works redefine their own rules from scratch. Thus, further research is needed to define the most effective rules;

- **Feature Extraction**: We analyzed all the feature extraction methods explored in the literature and observed a common trend toward eliminating the use of manual features. Thus, automatically learned features are becoming the standard in the field. However, a recent work [80] demonstrates that, in the binary similarity task, a small subset of manually crafted features is sufficient to obtain comparable performance. Therefore, we believe that further research is needed to investigate whether automatically extracted features can consistently outperform manual ones across all tasks and datasets;

- **Deep Learning Models**: We studied all the different deep learning models used and observed that most solutions, except for a few cases, utilize custom networks obtained from modifications to existing ones. We provide a systematization of the main types of modifications applied. From this analysis, we notice that these customizations often lead to complex and intricate architectures that are not often justified or compared with baseline models;

- **Pre-training Tasks**: For works relying on the pre-training and fine-tuning paradigm, we create a categorization of their pre-training tasks. While doing so, we noticed that newly introduced tasks are not often adequately compared against baseline tasks, such as the Masked Language Modeling (MLM) task. Additionally, with the exception of a few works [13, 91], the pre-training and fine-tuning paradigm is misused. In fact, most of the works use them only to solve a single task without testing their generalizability to different downstream tasks.

3.1 Goals

Although beeing a fundamental and a proficent area, there are no existing works that comprehensively organize the most recent literature. In 2019, [161] conducted a literature review on machine learning techniques for binary code analysis. More recently, in 2022, [105] delivered an in-depth evaluation of research employing deep learning for addressing binary similarity challenges. However, the binary similarity problem is not the unique problem that can be solved with deep learning-based solutions.

This thesis aims to bridge this gap by providing an exhaustive literature review focused on the application of deep learning approaches to binary analysis. Specifically, we started identifying the main building blocks for deep learning based binary analysis works: solved tasks 3.7, dataset 3.8, models 3.9 and pre-training tasks 3.10. For each of these main building blocks we tried to clusterize the identified literature into groups so as to identify common patterns or unexplored paths.

3.2 Primary Scope

We restricted the scope of our literature review to studies that address binary analysis tasks on benign executables, intentionally excluding literature related to malware analysis. The primary reason for this exclusion is that the field of analyzing complex malicious binaries presents specific challenges that differ significantly from those encountered with benign executables, such as intricate obfuscation and anti-debugging techniques, and security-sensitive datasets. These challenges merit a separate discussion. Conversely, simple malicious binaries are effectively comparable to benign executables for the tasks analyzed in our review. Additionally, we chose not to focus on the task of determining whether a binary is malicious, as this specific area has been extensively researched and documented in the literature with decades of research and numerous survey and Systematization of Knowledge (SoK) papers already published [100, 146].

3.3 Methodology

Our first step in our analysis has been to collected the recent relevant work in the literature. To achieve this, we identified works in the subject area by searching for specific keywords, such as "deep learning" and "binary analysis", within well known research databases such as *Google Scholar* [3] and *DBLP* [2]. Additionally, we expanded our research by also reviewing papers cited from these works or papers citing them. We discarded unpublished works, except when they were cited or used for comparisons by other relevant works. With this process we identified and analysed 54 research papers spanning 9 years of research (from 2015 till 2024).

3.4 Challenges of Deep Learning in the Binary analysis field

Deep learning techniques applied at the source code level have gathered considerable attention, with ongoing exploration and integration into commercial tools. For instance, Copilot [1], a code completion tool developed by Github and OpenAI, leverages these techniques to support software development. However, there are comparatively fewer solutions that apply such techniques to the domain of binary analysis. This is due to the intrinsic characteristics of binary code that introduce significant challenges, thus hindering the straightforward application of the techniques widely employed at the source code level:

- *Prevalence of compiler optimized code*: the compilation of source code at varying levels of optimization introduces a layer of complexity for analysts attempting to understand the code. Higher optimization levels can significantly obfuscate the code, making it challenging to understand. Techniques such as function inlining, aimed at enhancing execution speed, could merge functions with distinct semantics. This issue is further exacerbated by Link Time Optimization (LTO), which is designed to optimize a program across all its modules, thus leading to functions being inlined across source file boundaries. Therefore, creating a deep learning model capable of identifying complex logic within binaries demands an in-depth comprehension of their semantics;

- *Low level syntax*: source code is written with syntactically rich programming languages (C, C++, Java, etc.) that incorporate high-level construct such as loops, conditional statements, and high-level data structures. These elements are straightforward to identify and interpret within source code due to the clear, structured syntax of these languages. However, when translated into binary code, these constructs are reduced to a linear sequence of assembly instructions. For instance, both loops and conditional statements are represented in binary code through jump instructions. Distinguishing between these constructs requires a detailed analysis of the instructions surrounding the jumps, as well as an understanding of the specific patterns that signify loops versus conditional branches (see Figure 3.1). To make things worst, the conditions governing these jumps are often determined by the status of flags (RFLAGS registers in x86 architecture), which are indirectly set by prior instructions and thus not immediately apparent. In order to reconstruct the conditions, it is required to fully understand how instructions interact with and modfiy the processor's state over time. This contributes to further obscure the actual jump conditions which are fundamentals to fully understand the flow of control and the high-level logic of a binary program. Deep learning models should be meticulously designed to grasp the intricacies of low-level syntax along with its subtle nuances;

- *Lack, intricate or wrong symbolic information*: the process of stripping typically removes significant symbolic details from binary code, such as the names and types of variables, and the names of functions typically present in souce code. Even in those cases where such information is present it can be challenging

to understand. For example, certain programming languages (such as C++) employ name mangling; a technique that incorporates extra information within function names to manage function overloading or the use of identical identifiers in different scopes effectively. This method renders the names less intelligible to a human attempting to reverse-engineer the code. Furthermore, when dealing with highly optimized binaries, there is a risk that debug information—such as source line information and variable values—may be inaccurate [45, 92] . Building a model that depends on this potentially erroneous information for its training could lead to inaccuracies. Finally, while source code usually has comments inserted by developers to facilitate the code comprehension, binary code does not. Deep learning models need to comprehend the logic behind binary code snippets without relying on all such high-level hints;

- *Long distance dependencies*: instruction sequences in source code are typically more concise compared to their corresponding assembly sequences in binary code. In fact, in binary code instructions that are logically related may be separated by unrelated instructions. For example, in Figure 3.1a, the instructions 'mov r9, 0xd' and 'lea r13d, [rcx+r9]' are linked by a data dependency, even though they appear distantly within the code sequence. This dispersion of related instructions poses a significant challenge in identifying and understanding long-distance dependencies, which is a well known problem in NLP [167];

- *Sensitivity to changes*: binary code exhibits a higher sensitivity to modifications compared to source code. For example, altering all identifier names in the source code does not impact the underlying logic of the program. In contrast, minor adjustments in binary code can significantly alter or, in extreme cases, corrupt the semantics of the code snippets. Deep learning models should be adept at detecting these minute changes and understanding their implications on the program's overall semantics.

```
0:  mov   r9,0xd
7:  cmp   rax,rbx
a:  jle   17
10: inc   rax
13: add   rbp,0x3
17: add   rax,rcx
1a: lea   r13d,[rcx+r9]
```

(a) if-statement in assembly.

```
0:  cmp   rcx,0x0
4:  je    16
a:  dec   rcx
d:  add   rbp,0x3
11: jmp   0
16: add   rax,rcx
19: lea   r13d,[rcx+r9]
```

(b) loop in assembly.

Figure 3.1. Example of assembly snippets.

These factors not only necessitate more sophisticated approaches to understand binary code but also highlight the need for innovative solutions tailored to overcome the unique obstacles presented by binary formats.

3.5 Preliminary Definitions.

Before delving into the details of the literature review, let us first provide some preliminary definitions.

$B = \{b_1, b_2, \dots\}$	The set of all possible binaries.
$S = \{s_1, s_2, \dots\}$	The set of all possible source codes.
$CF = \{cf_1, cf_2, \dots\}$	The set of all possible compiler families and versions.
$OPT = \{opt_1, opt_2, \dots\}$	The set of all possible optimizations.
$ARCH = \{arch_1, arch_2, \dots\}$	The set of all possible architectures.
$OBF = \{obf_1, obf_2, \dots\}$	The set of all possible obfuscations.
$C \subseteq CF \times OPT \times ARCH \times OBF$	The set of all possible compiler configurations.
$F_b = \{f_b^1, \dots, f_b^n\}$	The set of all functions within a binary.
$BB_{f_b} = \{bb_{f_b}^1, bb_{f_b}^2, \dots\}$	The set of all basic blocks within a function.
$F_s = \{f_s^1, \dots, f_s^k\}$	The set of all functions within a source code.
$SS_{bb} = \{ss_{bb}^1, \dots, ss_{bb}^p\}$	The set of all strands of a basic block.
$R_{ss} = \{r_{ss}^1, \dots, r_{ss}^l\}$	The representative set of a strand.
$E_{f_b} \subseteq BB_{f_b} \times BB_{f_b}$	The set of edges between function's basic blocks.
$CFG_{f_b} = (BB_{f_b}, E_{f_b})$	The Control Flow Graph of a function.
$BB_b = \{bb_b^1, bb_b^2, \dots\}$	The set of all basic blocks within a binary.
$E_b \subseteq BB_b \times BB_b$	The set of edges between basic blocks of a binary.
$ICFG_b = (BB_b, E_b)$	The Interprocedural Control Flow Graph of a binary.

Table 3.1. Notation.

Binary program, functions, and basic blocks. Let B be the set of all possible binaries, and S be the set of all possible source codes of programs. Let $C \subseteq CF \times OPT \times OBF \times ARCH$ be the set of all possible compiler configurations, where CF, OPT, OBF and ARCH are the set of all possible compiler families, optimizations, obfuscation techniques, and target CPU architectures respectively. A compilation function is defined as:

$$\text{Compile} : S \times C \to B$$

For any source code $s \in S$ and compiler setting $c \in C$, the compilation function produces a binary $b \in B$:

$$b = \text{Compile}(s, c)$$

Each binary b contains a set of n binary functions $F_b = \{f_b^1, \dots, f_b^n\}$, while each function $f_b \in F_b$ contains a set of m basic blocks $BB_{f_b} = \{bb_{f_b}^1, \dots, bb_{f_b}^m\}$.

Each source code s contains a set of k source functions $F_s = \{f_s^1, \dots, f_s^k\}$.

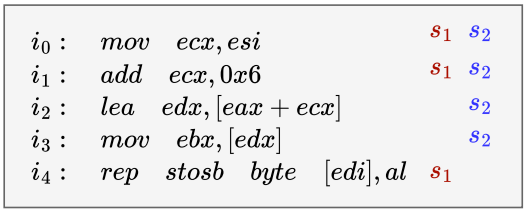


Figure 3.2. Example of strands in a CFG block.

Strands. A strand, as originally defined in [42], is a slice of a basic block constituted by all the instructions that are connected by def-use dependences. More specifically, consider a basic block as a sequence of assembly instruction $bb = [a_1, \ldots, a_r]$, it can be partitioned into p stands $SS_{bb} = \{ss^1_{bb}, \ldots, ss^p_{bb}\}$ such that $ss^i_{bb} \subseteq bb$ is a maximal subsequence of bb where all the instructions in ss^i_{bb} are connected by a def-use relationship. Moreover, it is worth noticing that strands are not disjoint, i.e. a given instruction can be part of multiple strands, i.e. $ss^i_{bb} \cap ss^j_{bb}$.

For instance, the basic block shown in Figure 3.2 consists of two strands. The strand s_1 includes instruction i_0, i_1, i_4, since instructions i_0 and i_1 modify the value of register `ecx` which is used by the `rep stosb` instruction to repeatedly place the content of al into the memory pointed by `rdi` for `ecx` times. The strand s_2 includes instruction i_0, i_1, i_2 and i_3 that contribute in defining the value of the `ebx` register.

Additionally, each strands can be represented by some output variables, such as register or a memory location values. Each of these output variables can be expressed symbolically as a functional representation of the input-output relationship of the strand. The collective set of such symbolic expressions constitutes the representative set of the strand $R_{ss} = \{r^1_{ss}, \ldots, r^l_{ss}\}$.

For instance, strand s_1 in Figure 3.2 has three output variables: the memory location pointed by `rdi` and the value of the registers `rdi` and `ecx`.

Control Flow Graph. Given a binary function f_b within a program $b \in B$, a $CFG_{f_b} = (BB_{f_b}, E_{f_b})$ is a directed graph representing the flow of control during function execution. Specifically, $E_{f_b} \subseteq BB_{f_b} \times BB_{f_b}$ is the set of edges, where each edge $e \in E_{f_b}$ is an ordered pair of nodes (bb_1, bb_2) within a function and representing a possible control flow transfer from the basic block bb_1 to bb_2 due to the presence of conditional or unconditional branches or other control flow transfer instructions.

Interprocedural Control Flow Graph. An Interprocedural Control Flow Graph (ICFG) is an extension of the traditional CFG since it covers multiple functions within a program. In particular, given a binary program $b \in B$, a $ICFG_b = (BB_b, E_b)$ is a directed graph representing the flow of control during program execution. Specifically, BB_b is the set of all basic blocks within a binary, and $E_b \subseteq BB_b \times BB_b$ is the set of edges. Each edge $e \in E_b$ is an ordered pair of nodes (bb_1, bb_2) representing a possible control flow transfer from the basic block bb_1 to bb_2 due to the presence of conditional or unconditional branches, function calls or other control flow transfer instructions.

Data Flow Graph. Given a binary program $b \in B$, a Data Flow Graph $DFG_b = (I_b, E_b)$ is a directed graph representing the data flow among instructions. Specifically, the graph nodes I_b are the set of all the instructions of the binary b and $E_b \subseteq I_b \times I_b$ is the set of edges. Each edge $e \in E_b$ is an ordered pair of nodes (i_1, i_2) representing a def-use relationship between instructions i_1 and i_2.

Call Graph. Given a binary program $b \in B$, a Call Graph $CG_b = (F_b, E_b)$ is a directed graph representing the calling relationships (callers and callees) among instructions. Specifically, the graph nodes F_b are the set of all the functions of the binary b and $E_b \subseteq F_b \times F_b$ is the set of edges. Each edge $e \in E_b$ is an ordered pair of nodes (f_b^1, f_b^2) representing a calling relationship between function f_b^1 and f_b^2 (i.e. f_b^1 is the caller and f_b^2 is the callee).

Abstract Syntax Tree. An Abstract Syntax Tree (AST) is a tree-based representation for the source code. In particular, inner nodes represent operators or control flow construct in the source code, while leaf nodes represent variables, constants or other data values. The AST representation is used for program analysis applications.

3.6 Deep Binary Analysis Pipeline

By analyzing the literature, we can derive a general binary analysis pipeline that is commonly used across all the works examined, typically with only minimal differences. This pipeline is represented in Figure 3.3. The remainder of this chapter is organized into sections that follow the stages of the aforementioned pipeline.

The first step of this pipeline is the dataset creation phase, which consists of several steps: raw dataset construction, binary representation extraction, preprocessing, and feature extraction. Each of these steps will be outlined in detail in Section 3.8. After extracting the dataset, specific deep learning models are used to solve a downstream task. Both models and tasks will be analysed in details in Section 3.9 and 3.7 respectively. Additionally, we will also analyse the pre-training strategies employed by works based on the Transformer network (Section 3.10).

3.7 Binary Analysis Downstream Tasks

Publications in the existing literature on binary analysis using deep learning and natural language processing techniques can be classified according to the different tasks they try to solve. In total, we have identified 25 different tasks. Figure 3.4 provides a visual representation of the binary analysis tasks solved in the revised literature. Specifically, we clusterized these tasks into 8 macro categories: Similarity (3.7.1), Toolchain Provenance (3.7.2), Disassembly (3.7.3), Decompilation (3.7.4), Debug Information Recovery and Repairing (3.7.5), Binary Code Understanding (3.7.6), Memory Usage (3.7.7) and Code Autorship (3.7.8).

Finally, it is worth mentioning that the relevant literature, with few exceptions [8,38,43,62,78,99,112,119,135], analyzes disassembled binaries, composed of sequences of assembly instructions, instead of raw binaries composed of sequences of bytes. We

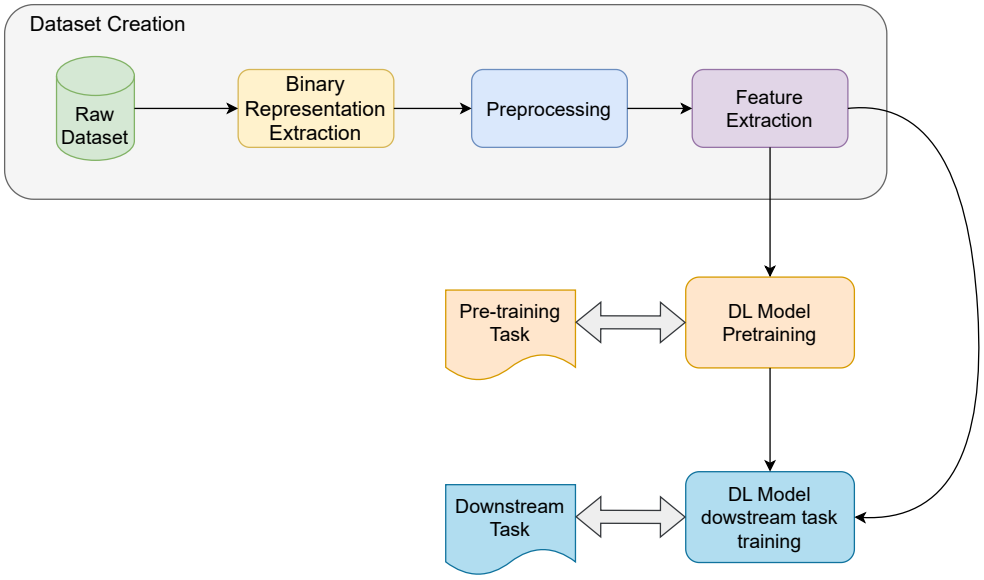


Figure 3.3. Deep Binary analysis pipeline.

will do the same, using the term binary to refer to a disassembled binary, explicitly stating when we are talking about raw binaries.

3.7.1 Similarity

Binary similarity consists of the systematic analysis of binary programs to determine their degree of resemblance, specifically aiming to identify and delineate the structural and semantic commonalities between them. Binary similarity is an important process in the cyber security context as it serves areas like malware detection, malware lineages, patch analysis, vulnerability detection, plagiarism detection, and so on [105]. Although function-level binary similarity is the most prevalent variant of binary similarity tasks, various other forms of this problem exist in the current literature. The following Sections will describe all such variants, including *Basic Block Similarity*, *Strand Similarity*, *Code Containment Problem*, *Binary Graph Alignment* and *Function-level Binary Souce Code matching*.

Function Similarity

The *Function Similarity* tasks consist of determining the similarity between two binary functions. Two binary functions are similar when they derive from the same source code compiled either with different compilers, optimization levels, obfuscation methods, or different architectures [160]. As it is defined, two functions that originate from different source code but have the same semantic are not considered similar. However, it has been shown [107] that solutions that have been trained on the purely syntactical definition generalises and cluster functions with the same semantic even when coming from different source codes. We identified 20 works solving the function similarity task: Gemini [160], VulSeeker [57], Zeek [133], $\alpha diff$ [99], GMN [93],

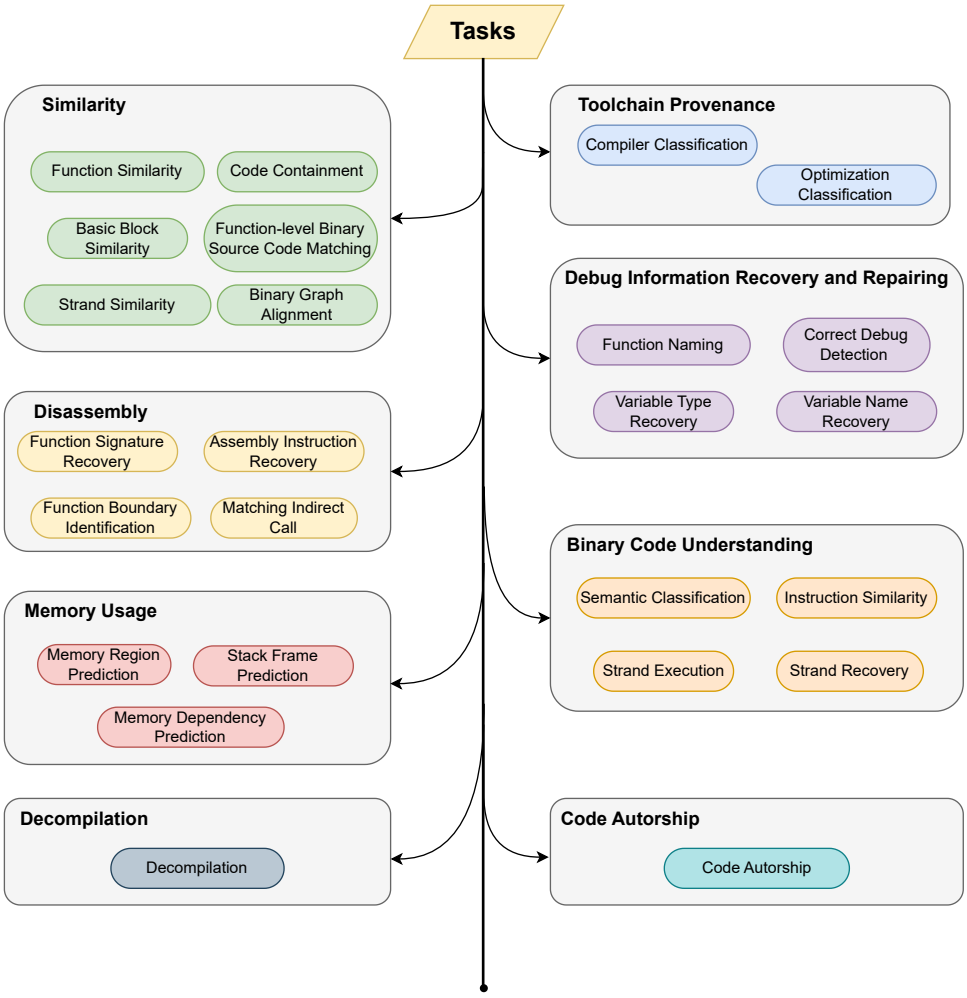


Figure 3.4. Tasks solved by state-of-the-art works.

Graphemb [106], SAFE [107], Asm2Vec [46], OrderMatters [168], TIKNIB [80], TREX [121], PalmTree [91], Binshot [4], JTrans [155], Sem2vec [154], Codee [163], VulHawk [103], BinFinder [127], BinBert [13], HermesSim [66].

Definition. Consider two binaries b_1 and b_2 such that $b_1 = \text{Compile}(s, c_1)$ and $b_2 = \text{Compile}(s, c_2)$ for two compiler settings c_1 and c_2. A function $f^1_{b_1} \in F_{b_1}$ is similar to a function $f^2_{b_2} \in F_{b_2}$ if and only if both functions derive from the same function f_s in the original source code s. Otherwise, the two functions are dissimilar.

The *Function Similarity* task consists in determining whether a given function f_1 is similar or not to a given function f_2. Said otherwise, the task is to learn the indicator function $\text{Sim} : F_{b_i} \times F_{b_j} \to \{0, 1\}$ that outputs one when the functions are similar and zero otherwise.

In order to test the effectiveness of the model, some works use a variant of the function similarity task named *Function Search*, which reflects better real-world usage.

Function Search. In the *function Search* task, given a database of functions and a set of query functions, the network must identify, for each query, the similar functions in the dataset. Works solving this problem are: Gemini [160], VulSeeker [57], Zeek [133], $\alpha diff$ [99], SAFE [107], Asm2Vec [46], OrderMatters [168], TREX [121], Binshot [4], JTrans [155], Sem2vec [154], Codee [163], VulHawk [103], BinFinder [127], BinBert [13], HermesSim [66].

Definition. Given a query function q, the *Function Search* task consists in retrieving, from a large database D of functions, those that are similar.

Basic Block Similarity

The *Basic Block Similarity* task consists of determining whether two basic blocks derive from the same piece of source code compiled either with different compilers, optimization levels, obfuscation methods, or different architectures. Works solving this problem are: InnerEye [175], XArchInstrEmb [128] and PalmTree [91].

Definition. Consider two binaries b_1 and b_2 such that $b_1 = \text{Compile}(s, c_1)$ and $b_2 = \text{Compile}(s, c_2)$ for two compiler settings c_1 and c_2.

A basic block $bb_{f_{b_1}} \in BB_{f_{b_1}}$ is similar to a $bb_{f_{b_2}} \in BB_{f_{b_2}}$ if and only if $bb_{f_{b_1}}$ and $bb_{f_{b_2}}$ derive from the same piece of the source code s.

The *Basic Block Similarity* task consists in determining whether a given basic block bb_1 is similar or not to a given basic block bb_2. Said otherwise, the task is to learn the indicator function $\text{Sim} : BB_{b_i} \times BB_{b_j} \to \{0, 1\}$ that outputs one when the blocks are similar and zero otherwise.

Strand Similarity

The *Strand Similarity* task consists of determining whether two strands are similar, i.e. they have an overlapping semantic (non-empty intersection of the representative

sets). In contrast to function and basic block similarity tasks, which define similarity based on whether two pieces of assembly code originate from the same source code segment, the strand similarity task focuses on the values computed by specific assembly code portions. Therefore, it necessitates a semantic understanding of the assembly code in question. This task is solved only by BinBert [13].

Definition. Given two strands ss_{bb_1} and ss_{bb_2} with their corresponding representative sets $R_{ss_{bb_1}}$ and $R_{ss_{bb_2}}$, they are said to be similar if and only if $R_{ss_{bb_1}} \cap R_{ss_{bb_2}} \neq \emptyset$

The *Strand Similarity* task consists in determining whether a given strand ss_{bb_1} is similar or not to a given strands ss_{bb_2}. Said otherwise, the task is to learn the indicator function $\text{Sim} : SS_{bb_i} \times SS_{bb_j} \rightarrow \{0, 1\}$ that outputs one when the strands are similar and zero otherwise.

Code Containment Problem

The *Code Containment* problem consists of determining whether a piece of binary code is contained within another piece of binary code compiled either with different compilers, optimization levels, obfuscation methods, or different architectures. This task is solved only by InnerEye [175].

Definition. Consider two binaries b_1 and b_2 such that $b_1 = \text{Compile}(s, c_1)$ and $b_2 = \text{Compile}(s, c_2)$ for two compiler settings c_1 and c_2 and two pieces of binary code $p_{f_{b_1}} \in P_{f_{b_1}}$ and $p_{f_{b_2}} P_{f_{b_2}}$ representing paths in $CFG_{f_{b_1}}$ and $CFG_{f_{b_2}}$ respectively.

The *Code Containment* task consists in determining whether a pieces of binary code $p_{f_{b_1}}$ is contained within another piece of binary code $p_{f_{b_2}}$. Said otherwise, the task is to learn the indicator function $\text{Contain} : P_{f_{b_i}} \times P_{f_{b_j}} \rightarrow \{0, 1\}$ that outputs one when the first piece of binary code is contained within a second piece of binary code and zero otherwise.

Binary Graph Alignment

Similarly to the definition provided by [49], the *Binary Graph Alignment* tasks consist of assessing the similarities and delineating the differences between two binary programs compiled either with different compilers, optimization levels, obfuscation methods, or different architectures by identifying possible matches among their basic blocks. This task is solved by DeepBinDiff [49] and XBA [81].

Definition. Consider two binaries b_1 and b_2 such that $b_1 = \text{Compile}(s, c_1)$ and $b_2 = \text{Compile}(s, c_2)$ for two compiler settings c_1 and c_2, their corresponding ICFGs, $ICFG_{b_1} = (BB_{b_1}, E_{b_1})$ and $ICFG_{b_2} = (BB_{b_2}, E_{b_2})$ and the indicator function Sim defined for the basic block similarity.

The *Binary Graph Alignment* task consists in finding the optimal basic block matching that maximizes the similarity between $b1$ and $b2$:

$$Sim(b1, b2) = \max_{m_1,\ldots,m_k \in M(b1,b2)} \sum_{i=1}^{k} Sim(m_i)$$

where $M(b1, b2) \subseteq (BB_{b_1} \times BB_{b_2})$ represent a set of matching pairs $(bb^i_{b_1}, bb^j_{b_2})$.

Function-level Binary Source Code Matching.

The *Function-level Binary Source Code Matching* problem consists of determining whether a binary function matches a source code function, i.e. it derives from the compilation of that source code function. This task is solved only by CodeCMR [169].

Definition. Consider a source code s and a binary b such that $b = \text{Compile}(s, c)$ for a compiler setting c. A function $f_b \in F_b$ match to a function $f_s \in F_s$ if and only if f_b derives from the function f_s in the original source code s. Otherwise, the two functions do not match.

The *Function-level Binary Source Code Matching* task consists in determining whether a given binary function f_b match or not to a given source code function f_s. Said otherwise, the task is to learn the matching function $\text{Match} : F_b \times F_s \to \{0, 1\}$ that outputs one when the functions match and zero otherwise.

3.7.2 Toolchain Provenance

Toolchain Provenance aims at identifying tools and configurations used to produce a given executable. It is especially helpful for digital forensics investigations as it can pinpoint the environment in which binaries and especially malware have been compiled [112]. Additionally, toolchain provenance is crucial for determining security flaws possibly inserted by specific vulnerable compiler versions inside binary code [48]. Since such information is usually omitted from binaries in production environments, recovering it becomes crucial to ensure the integrity and security of the software. Toolchain provenance tasks require the model to understand the specific instruction patterns created by compilers to identify the toolchain.

Compiler Classification.

The *Compiler Classification* task involves determining the particular compiler family (eventually with version) used to compile a given binary code. This task is solved by GraphEmb [106], o-glassesX [112] and BinBert [13].

Definition. Given a function f_b (or a binary b), the *Compiler Classification* is to determine the specific compiler family (eventually with version) $cf \in CF$ used to compile f_b (or the binary b).

Optimization Classification.

The *Optimization Classification* task involves determining the particular compiler optimization used to compile a given binary code. This task is solved by GraphEmb [106], Himalia [36], OrderMatters [168], o-glassesX [112] and BinBert [13].

Definition. Given a function f_b (or a binary b), the *Optimization Classification* is to determine the specific optimization $opt \in OPT$ used to compile f_b (or the binary b).

3.7.3 Disassembly

The *Disassembly* task is performed by several Commercial off-the-shelf disassemblers. Such disassemblers are available in both static and dynamic variants. Static disassemblers, such as the simpler linear disassembler *objdump*, decode bytes in a sequential manner. More advanced ones, like recursive disassemblers (e.g. IDA), follow the control flow graph to decode bytes so as to avoid inline data typically introduced by commercial compilers. However, these may miss some blocks due to indirect jumps or calls. Such issues are exhacerbated in x86 compilers where innstructions have varying lenght thus leading to the possible presence of complex construct like overlapping and misaligned instructions which can cause troubles while disassembly [9]. Consequently, dynamic disassemblers have emerged, leveraging execution data to more accurately decode bytes and address the limitations of static methods. Unfortunately, dynamic disassemblers strongly depend on the code coverage, since they can decode only the bytes of executed instructions. Additionally, dynamic disassemblers can be slower and more resource intensive than traditional static disassemblers. To overcome all such limitations and improve the accuracy of such disassemblers, the research community started to investigate several techniques including deep-learning based solutions [38, 119, 135, 166].

Specifically, following the definition used by [9,119], the disassembly task involves recovering the following disassembly primitives:

- assembly-level instructions;
- function boundaries, the end and start addresses of functions within a binary;
- function signatures, the list of function parameters;
- Interprocedural Control Flow Graph.

Thus, based on this definition, we categhorized the revised literature regarding disassembly within the following subtasks: *Assembly Instruction Recovery*, *Function Boundary Identification*, *Function Signature Recovery* and *Matching Indirect Calls*. Although the latter does not directly contribute to building the ICFG, it plays a crucial role by recovering key information necessary for constructing a comprehensive ICFG—specifically, resolving indirect calls.

Assembly Instruction Recovery

The task of *Recovering Assembly Instructions* consists of identifying the bounds of each individual assembly instruction within the code section of a stripped binary. The challenge lies in accurately differentiating and isolating these instructions from the binary's continuous byte stream, especially considering the variability in instruction lengths and the presence of inline data within executable code segments. This task requires a semantic understanding of the code, especially because it is necessary to also understand the bytes targeted by indirect jumps and calls. Works solving this problem are XDA [119] and DeepDi [166].

Definition. Given a binary b consisting of series of byte sequences, the goal of the *Assembly Instruction Recovery* task is to recover a set of assembly instructions start and end address pairs (i.e. delineate the boundaries of each instruction):

$$A_b = \{(a_s^1, a_e^1), \ldots, (a_s^m, a_e^m)\}$$

where a_s^i and a_e^i are the start and the end address of the i^{th} instruction.

Function Boundaries Identification

Similarly to the definition of [135] and [19], the *Function Boundary Recovery* task is the process of delineating the start and end points of functions within a binary executable. Among the works solving this problem we have identified RFBNN [135], XDA [119] and DeepDi [166].

Definition. Given a binary b composed of n functions $F_b = \{f_b^1, \ldots, f_b^n\}$, the goal of the *Function Boundaries Identification* task is to recover a set of function start and end address pairs:

$$A_b = \{(a_s^1, a_e^1), (a_s^2, a_e^2), \ldots, (a_s^k, a_k^1)\}$$

where a_s^j and a_e^j are the addresses of the first and last byte of the function f_b^i respectively.

Function Signature Recovery

The *Function Signature Recovery* problem involves identifying the number and types of arguments of a function within a stripped binary. This task is solved by PalmTree [91] and Eklavya [38].

Definition. Given a binary function $f_b \in F_b$, where b is a stripped binary, the goal of the *Function Signature Recovery* task is to recover the number n of arguments and for each argument determine its specific type $t \in T$.

Matching Indirect Calls

Constructing a comprehensive ICFG poses significant challenges, primarily because of the existence of indirect jumps or calls, where the destination address of an

instruction is determined only during runtime. To enhance the completeness of the ICFG, various techniques based on fuzzing have been investigated [25]. More recently, Calle [174] has begun to explore the application of deep learning to identify the specific function invoked by a call instruction, thus further improving the accuracy and depth of the ICFG. This approach holds promise for addressing the complexities associated with indirect jumps and advancing the state-of-the-art in control flow analysis.

Definition. Consider a function f_b^1 within a binary progam b and a call-site $cs \in CS_b$ within a function f_b^2 in the same binary b. The *Matching Indirect Call* task is to learn a matching function $Match : F_b \times CS_b \rightarrow \{0, 1\}$ that output 1 if the function signature of f_b^1 matches the call-site cs and 0 otherwise.

3.7.4 Decompilation

Decompilation involves transforming binary code back to its higher-level representation. Unlike disassemblers, which translate binary into assembly code, decompilers advance this process by generating a more comprehensible form of code that allows for easier analysis and interpretation by human analysts. Unfortunately, high-level information is usually lost during compilation, thus recovering the original source code from its binary representation is an hard task [102]. In fact, the code resulting from decompilation is usually not the same as its original source code but rather an approximation.

Numerous commercial decompilers are available today, including IDA Hex-Ray [1], Ghidra Decompiler [2], and Boomerang [3]. However, these tools often produce incomplete or inaccurate code [50, 102]. Some recent works, like RNND [78], CODA [55] and NeurDP [29], started exploring the use of neural networks to solve this problem.

Definition. Consider a binary program b and a source code s such that $b = \text{Compile}(s, c)$ for a compiler setting c. An high-level code $d \in S$ is the decompiled code of the binary b if d is semantically equivalent to the original source code s.

The *Decompilation* task is to learn a function Decompile : $B \rightarrow S$, that, given a binary code b produce high-level code $d = Decompile(b)$ such that s and d produce the same output for every possible input values.

3.7.5 Debug Information Recovery and Reparing

Debug symbols are generated by compiler programs and typically include information about functions and variables, such as name, location, type, and size which is helpful for debugging and security analysis of a binary. For ELF binaries, debug information is stored within a specific executable section in the DWARF format [51], while debug

[1] https://hex-rays.com/decompiler/
[2] https://github.com/NationalSecurityAgency/ghidra
[3] https://boomerang.sourceforge.net/

information of Windows PE binaries resides in a separate dedicated file known as a Program Database (PDB).

Since this information is not required for execution, debug symbols are usually removed from binaries. The process of removing such information is known as stripping. The primary motivation for stripping binaries is to enhance performance; removing debug information reduces overhead and file size. Additionally, commercial vendors often strip binaries to safeguard their intellectual property, and malicious actors do so to complicate the analysis of their malware. For these reasons, some works have started to investigate the applicability of deep-learning-based solutions for *Function Naming* [41, 53, 56, 76, 82, 117] and *Variable Names and Types Recovery* [18, 32, 84, 110, 122].

Additionally, some recent works demonstrate that, in case of highly optimized binaries, debug information—such as source line information and variable values—may be inaccurate [45, 92]. Such inaccuracies can be generated either during the compilation chain by the compiler or later at debugging time by the debugger itself. One of the work presented in this thesis (Chapter 4), started to test the applicability of deep-learning techniques to detect such bugs [12].

Function Naming

The *Function Naming* problem consists of assigning a string to a binary function. Such string should represents a meaningful name for that function, i.e. a name that an expert programmer would assign and that captures the semantics and the role of the function inside the software [11]. Assigning a name to a function requires a deep understanding of the code semantics. This problem have been studied by NERO [41], Punstrip [117], NFRE [56], SymLM [76], XFL [53] and AsmDepictor [82].

Definition. Given a binary function $f_b \in F_b$, where b is a stripped binary, the goal is to find a meaningful name $fn \in FN$ for a specific function. A meaningful name is a name that an expert programmer would assign and that captures the semantics and the role of the function inside the software.

Said otherwise, the *Function Naming* task is to learn a renaming function $Rename : F_b \rightarrow FN$, that assign a function name $fn = Rename(f_b)$ to a function f_b.

Variable Names Recovery

The *Variable Name Recovery* task consists of assigning meaningful names to variables contained inside a decompiled binary code. As for the function naming task, a meaningful name for a variable is a name that a programmer would assign to that variable and that reflect its role and semantic inside the software. Works solving this problem are: DIRE [84], DIRECT [110], VarBert [18] and DIRTY [32].

Definition. Given a decompiled binary $d_b \in S$ with n variables V, where b is a stripped binary, the goal is to find a meaningful name $vn \in VN$ for a specific variable $v \in V$. A meaningful name is a name that an expert programmer would assign and that captures the semantics and the role of the variable inside the software.

Said otherwise, the *Variable Name Recovery* task is to learn a renaming function $Rename : D_b \times V \rightarrow VN$, that assign a variable name $vn = Rename(d_b, v) \quad \forall v \in V$.

Variable Types Recovery

The *Variable Types Recovery* task consists of recovering the types of variables contained inside a decompiled binary code. The work solving this task is DIRTY [32].

Definition. Given a decompiled binary $d_b \in S$ with n variables V, where b is a stripped binary, the goal is to recover the correct type $t \in T$ for a specific variable $v \in V$.

Said otherwise, the *Variable Type Recovery* task is to learn a retyping function $Retype : D_b \times V \rightarrow T$, that assign the correct type $t = Retype(d_b, v) \quad \forall v \in V$.

A variant of this task is solved by Stateformer [122]. Specifically, Stateformer recover variable types directly from the binary code instead of the decompiled one.

Correct Debug Detection Problem

A pair compiler/debugger can be seen as a toolchain function TC that maps each source code s to a debug trace $T = TC(s)$, which is a an ordered list of elements, each representing a *step* over a machine instruction. A perfect toolchain TC should never generate a bugged trace. The *Correct Debug Detection* problem aims at finding bugs in the toolchain, and so to find instances of programs that generate wrong debug traces under TC. In this regards, the set $Traces$ of all possible traces generated by TC for all valid programs [4] can be partitioned into the set of correct traces $NoBug$ and the one of bugged traces Bug. This task is solved only by our work Neuro-Debug^2 [12].

Definition. Given a trace $T \in Traces$ the *Correct Debug Detection* problem gives as output 0 if T is a correct debug trace ($T \in NoBug$) and 1 if T is a bugged debug trace ($T \in Bug$).

3.7.6 Binary Code Understanding Tasks

Recovering semantic information from binaries is a useful feature for reverse engineering. In fact, understanding the functionalities of binaries helps human reverse engineers in reasoning about the software's purpose thus leading to quicker understanding of complex software behaviors. Additionally, semantic tasks can be used as benchmarks for assessing the effectiveness of deep learning models in understanding binary snippets.

[4] A program is valid if can be compiled by the toolchain and is free of undefined behaviors.

Semantic classification

The *Semantic Classification* task involves assigning a semantic class to each binary function to identify its general semantic behavior. One of the difficulties of this task lies in the dataset construction phase, since it requires manually assigned labels. This task has been solved only by SAFE [107].

Definition. Given a binary function $f_b \in F_b$, the semantic classification task is to learn a function $AssignSemantic : F_b \rightarrow SC$, that assign a semantic class $sc \in SC$ to a binary function f_b.

Instruction Similarity Task

The *Instruction Similarity Task* involves assessing the similarity between two assembly instructions, which is primarily determined by their semantic meaning. Instructions are categorized into semantic classes based on their opcodes, grouping together instructions with similar semantic meanings. An example of such categorization can be found in the x86 Assembly Language Reference Manual [5]. For instance, the `add` and `sub` instructions are classified as similar because they both perform mathematical operations on two values. The *Instruction Similarity Task* is solved by XArchInstrEmb [128], PalmTree [91], MAIE [156] and BinBert [13].

Definition. Given the set all the possible instructions $I = \{i_1, i_2, \ldots, i_n\}$, it can be partitioned into semantic classes $SC = \{sc_1, sc_2, \ldots, sc_m\}$, where each class is defined based on opcode characteristics. Two instructions $i_1 \in I$ and $i_2 \in I$ are similar if and only if they belong to the same semantic class sc_i, i.e. $i_1 \in sc_i$ and $i_2 \in sc_i$.

The *Instruction Similarity* task is to learn a function $Sim : I \times I \rightarrow 0, 1$ that given two instructions $i_1 \in I$ and $i_2 \in I$ outputs 1 if they are similar and 0 otherwise.

It is worth noting that some of the analyzed papers focus on solving two related variants of this task: the *Opcode Outlier Detection* task [13, 91] and the *Nearest Neighbor Istruction* task [156].

Opcode Outlier Detection Task. The *Opcode Outlier Detection* task has been defined by PalmTree [91] and solved also by our work BinBert [13]. This task involves analyzing a set of n instructions where $n - 1$ belong to the same semantic class, and one is an outlier. The objective is to identify the outlier instruction among the set of n instructions.

[5] https://docs.oracle.com/cd/E26502_01/html/E28388/ennbz.html

Definition. Given:

- A set of n instructions $I = \{i_1, i_2, \ldots, i_n\}$
- A corresponding set of semantic classes $SC = \{sc_1, sc_2, \ldots, sc_m\}$, where each class is defined based on opcode characteristics.

The objective is to identify the outlier instruction $i_o \in I$ such that i_o belongs to a different semantic class from the rest of the instructions in I.

Formally, given a function $f : I \to SC$ that assigns each instruction to its semantic class, the *Opcode Outlier Detection* task is to find i_o for which:

$$f(i_o) \neq f(i_k), \forall i_k \in I \setminus \{i_o\}$$

Nearest Neighbor Istruction Task. The *Nearest Neighbor Instruction* task, as utilized by MAIE [156], assesses the model's comprehension of instruction semantics within a multi-architecture environment.

Definition. Given a query instruction q belonging to a certain ISA, the *Nearest Neighbor Instruction* is to recover the top-K similar instructions in other ISAs.

Strand Execution

The *Strand Execution* task has been firstly introduced by our work BinBert [13]. The goal of this task is to test the network ability of understanding the execution behaviour of a binary code snippet, specifically a strand.

Definition. Given a strand ss and the set of all possible inputs I and output values O, the *Strand Execution* task is to learn a function $Compute : SS \times I \to O$, that given an input vector $i \in I$ representing values for all the input variables of a strand, produce an output vector $o = compute(ss, i)$ representing the output values for all the possible output variables of a strand.

Strand Recovery

The *Strand Recovery* tasks has been firstly introduced by by our work BinBert [13]. The goal of this task is to recover strands from the assembly instructions composing a basic blocks. In particular, this task test the network ability of understanding the semantic of binary code snippets (i.e. strands), since it has to infer the dependencies among instructions, including those created by implicit registers such as RFLAGS.

Definition. Given a basic block $bb \in BB$, the *Strand Recovery* task is to find the set of its p strands $SS_{bb} = ss^1_{bb}, \ldots, ss^p_{bb}$.

3.7.7 Memory Usage

Binary memory analysis is a critical task, particularly for post-mortem program analysis. Identifying the precise cause of program crashes resulting from erroneous input values is crucial for detecting memory corruption vulnerabilities [62]. Various

techniques, such as hardware tracing [159], have been explored to trace the sequence of instructions leading to a crash. However, these techniques often struggle to capture the complete trace, making it challenging to reconstruct the crashed memory state and, consequently, the entire control flow. Specifically, significant efforts have been dedicated to identifying memory aliases, which represent references to the same memory location and that can help in identifying the crash cause. For instance, Value Set Analysis (VSA) [16] aims to analyze and track the possible values that variables or memory locations can hold during program execution by approximating the set of possible values at each program point, thus helping in identifiying wether two variables point to the same memory location. Motivated by this, some works started investigating the possibility of using deep learning techniques to enhance VSA accuracy (*Memory Region Prediction* task [62]) or to directly identify memory aliases (*Memory Dependency Prediction* task [120]).

Furthermore, patching a binary program without recompiling it is a common and essential task. To accomplish this, understanding the stack layout of the function to be patched is crucial. Recent studies have explored the application of deep learning techniques to determine the size of each function's stack frame in a program, a process referred to as *Stack Frame Size Recovery* [43].

Memory Region Prediction

The *Memory Region Prediction* task is used to improve the accuracy of VSA and consists of predicting the memory region accessed by instructions within a binary code trace. This task has been solved by DeepVSA [62] and PalmTree [91].

Definition. Consider an execution trace T of a binary program b consisting of a sequence of n assembly instructions $I = (i_1, \ldots, i_n)$ and the set of all possible memory regions $M = \{stack, heap, global, other\}$.

The *Memory Region Prediction* task is to learn a function $MemPredict : T \times I \rightarrow M$ that assign a memory region $m \in M$ to each instruction i_i in the trace T.

Memory Dependency Prediction

The *Memory Dependency Prediction* task consists of determining wether two assembly instructions within a binary program trace refer to the same memory location. This task has been solved only by NeurDP [29].

Definition. Consider an execution trace T of a binary program b consisting of a sequence of n assembly instructions $I = (i_1, \ldots, i_n)$.

The *Memory Dependency Prediction* task is to learn a function $DepPredict : T \times I \times I \rightarrow 0, 1$, that given a trace T and an instruction pair i_i and i_j output 1 when they refers to the same memory location and 0 otherwise.

Stack Frame Prediction

The *Stack Frame Prediction* task is to determine the maximum size of the stack frame of a given function. This task has been solved only by StackBert [43].

Definition. Consider a function f_b of a binary program b. The *Stack Frame Prediction* task is to learn a function $StackPredict : F_b \rightarrow \mathbb{Z}$ that given a function f_b outputs the maximum size of its stack frame $size = StackPredict(f_b)$.

3.7.8 Code Autorship

The problem of *Code Authorship* involves identifying the authors of a particular binary program. This task is valuable for detecting software plagiarism, conducting malware analysis, and digital forensics investigations. While some may question whether the author's coding style can survive the compilation process, recent studies [108, 131] demonstrate that the author's style is preserved in compiled code, even resisting optimizations and certain obfuscation techniques [71]. An interesting aspect is that all existing approaches to this problem are based on the *Closed World Assumption* (CWA) [60], which assumes a finite set of possible authors. This assumption is limiting and distant from real scenarios, especially in cases involving malware, where new authors emerge over time. Another factor to consider is that a single binary program is often written by multiple developers, each potentially contributing with a different coding style. Consequently, multiple authors may be associated with a single binary. Recent approaches have explored the applicability of deep learning techniques to the code autorship problem on benign executables: BinEye [8] and BinMLM [137].

Definition. Given a binary code b, the *Code Autorship* task is to learn a function $AuthorAssign : B \times A \rightarrow 0, 1$ which output 1 if a given author $a \in A$ have contributed to write the binary code b and 0 otherwise.

Observed Gaps in the Binary Tasks

From our analysis of binary tasks, we identified two main gaps:

- Unexplored tasks: The majority of the reviewed works focus on the function similarity task, while many other tasks remain largely unexplored. In fact, some tasks have been addressed by only a single study;
- Inconsistent task definitions: Tasks that are essentially equivalent are often defined in slightly different ways, making it challenging or even impossible to compare solutions.

3.8 Dataset

In this section, we discuss the articles we reviewed from the perspective of the datasets used, encompassing the data generation methodology. We take a holistic approach and analyze the entire process, starting from the selection of binaries, which serve as the raw data for the models we studied (this is done in the Raw Dataset Section 3.8.1), to how a raw binary is represented by the deep learning model. For instance, the binary could be converted into a graph-based or linear representation (this is done in the Binary Representation Section 3.8.2). We also

examine how different papers process the basic operations that compose the binary (this is done in the Preprocessing and Tokenization Section 3.8.3).

To clarify, by *binary representation*, we refer to the way in which the sequence of basic operations composing the binary is represented. This is distinct from how individual basic operations are represented. For example, one could represent the entire binary using its Interprocedural Control Flow Graph (ICFG) and then represent the operations inside each basic block either by using the block's binary code (i.e., the hexadecimal code) or by employing a bag-of-instructions approach, counting how many instructions fall into certain categories. Finally, in the Feature Extraction Section 3.8.4, we discuss how the papers extract the actual features, i.e., the vectors of numbers on which the DNNs operate, from the binary representation described above.

3.8.1 Raw Dataset

All the datasets reviewed in the literature, although designed for specific tasks, require an initial phase of collecting raw executables. In this thesis, we will refer to these executables as the *raw dataset*. Figure 3.5 presents an idealized pipeline for *raw dataset* collection. This pipeline illustrates the general steps typically involved in creating the *raw dataset*. It is important to note that not all the works we surveyed utilized every step in this pipeline.

The pipeline has two collection points. At the first collection point, the source code of projects is gathered and compiled into binaries, which then undergo a filtering process that will be detailed later. The second collection point is where precompiled binaries are introduced into the pipeline and proceed directly to the preprocessing step. This dual-input model accommodates both studies that utilize source code and those that use precompiled binaries.

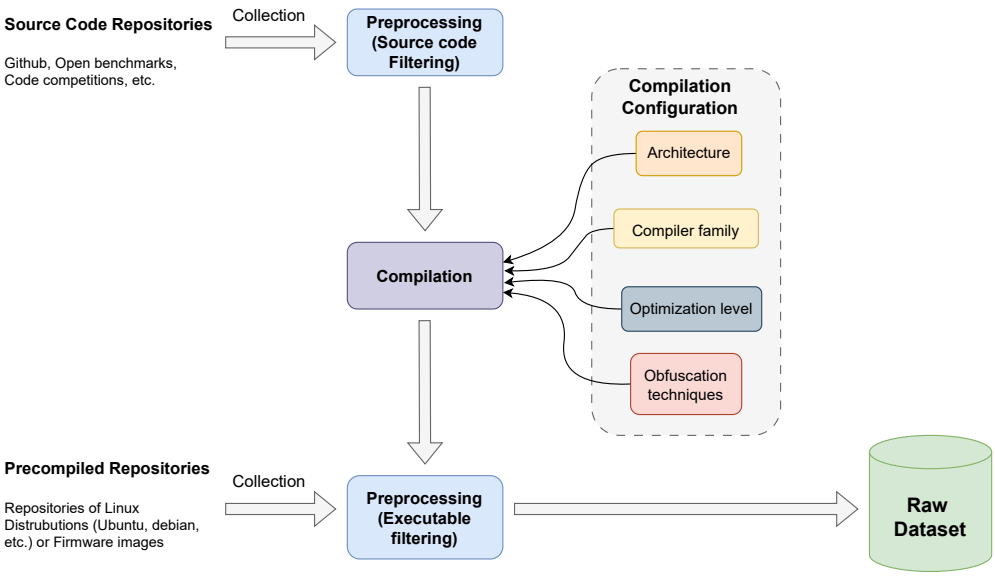


Figure 3.5. Raw dataset creation pipeline.

Pre-compiled vs Source Code. From our analysis of the literature, we found that the majority of works opted to compile open source projects themselves (See Table 3.2), either collected from open source code repositories such as Github or collected from publicly available benchmarks such as those of the *Standard Performance Evaluation Corporation* (SPEC) designed to test computer performances. Only a minority of studies opted for pre-compiled projects, such as repositories from Linux distributions [53, 56]. Some other works used both precompiled binaries and binaries compiled from scratch [99, 106, 117, 174], while others utilized precompiled firmware images from specific IoT vendors to test the model's ability to identify vulnerable binaries [57, 103, 121, 160].

Regarding the selection of source code projects compiled from scratch, it is worth noting that these are often chosen from a common pool, such as open-source projects like OpenSSL, binutils, findutils, and others. A consistent trend observed across all analyzed studies, even for similar tasks, is the creation of custom raw datasets without thorough validation of their overlap with prior research. As a result, even if many studies rely on source code datasets that are ostensibly identical, the obtained raw data can differ due to variations in the implementation of compilation and preprocessing steps. This makes impossible to compare the majority of results in the current literature.

From our analysis, only a few studies reused the same raw datasets as others. For instance, VarBERT [18] and DIRECT [110] used the same dataset as DIRE [84] for the task of variable name recovery, [105] and BinFinder [127] used subsets of the datasets from TREX [121] and TIKNIB [80] respectively, while HermesSim [66] used one dataset from [105].

Another notable point is that, the majority of the binaries used in the studies are in ELF format, while only few studies include PE files for the Windows OS [8, 81, 119, 135, 166].

Source Code Pre-Processing. After collecting source code projects, some works perform a preprocessing step finalized at removing some redundacies at the source code level. For instance, NERO [41], remove projects that compile with static linking so as to ensure that no dependencies are inserted inside the executables and also remove different versions of the same project. JTrans [155] remove all non C/C++ projects by looking at specific keywords in the PKGBUILD file.

Compilation. After preprocessing, the source code needs to be compiled. This involves defining the compilation configurations, including the platform, compiler family, compiler optimizations, and whether or not to use obfuscation strategies. The prevailing trend in the literature is to use a compilation step that employs different compiler families (e.g., GCC, clang, ICC) with varying optimization levels (e.g., O0, O1, O2, O3). This approach allows the same source code to be compiled into several significantly different binaries. Most of the works rely on the x86 architecture, while some others use cross-compilation to generate binaries for different ISAs (e.g., ARM, MIPS, PPC). A few studies also incorporate obfuscation techniques into the compilation process, typically using the LLVM obfuscator.

Compiled Binaries Filtering. After compilation, some works perform a preprocessing phase to filter out some executables. For instance NERO [41] remove all executables that are suspected to be test or examples and also executables coming from non C code.

After the executable filtering phase, it is possible to obtain the raw dataset.

Year	Work	Sources	#Binaries	ISA	Toolchain	Optimizations	Obfuscation	OS
2015	RFBNN [135]	binutils, coreutils, findutils and binaries from open-source projects for Windows	2200	x86, x86-64	gcc, icc	O0, O1, O2, O3	None	Linux, Windows
2017	Gemini [160]	OpenSSL	18269	x86-64, aarch64, MIPS64	gcc-5	O0, O1, O2, O3	None	Linux
2017	Eklavya [38]	binutils, diffutils, coreutils, findutils, inetutils, sg3utils, usbutils, util-linux	2000	x86, x86-64	clang, gcc	O0, O1, O2, O3	None	Linux
2018	α-Diff [99]	31 projects from GitHub repositories	69989 *	x86-64	clang-3 (2), gcc-4 (2), gcc-5 (1)	NA	None	Linux
2018	RNND [78]	Fedora selections for the MUSE project	NA	x86-64	clang-3	O0	None	Linux
2018	VulSeeker [57]	BusyBox, Coreutils, OpenSSL	NA	x86, x86-64, ARM, aarch64, MIPS, MIPS64	gcc-4, gcc-5	O0, O1, O2, O3	None	Linux
2018	Zeek [133]	Apache Mesos, bash, binutils, bzip2, coreutils, cURL, FFmpeg, Git, httpd, ntp, OpenSSL, QEMU, Snort, tar, util-linux, wget, Wireshark	20680 *	x86-64, aarch64	clang-3 (4), clang-4 (1), gcc-4 (3), icc-14, icc-15	O0, O1, O2, O3	None	Linux
2019	Asm2Vec [46]	BusyBox, coreutils, ImageMagick, libgmp, libcurl, libtomcrypt, OpenSSL, PuTTYgen, SQLite, zlib	68 *	x86-64	clang-3 (2), gcc-4, gcc-5, icc (2)	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB)	Linux
2019	BinEye [8]	GitHub projects, code from Google Code Jam competition and some malware	31150	x86-64	clang, gcc, g++, MVSC 2010	NA	O-LLVM (BCF, FLA, SUB)	Windows
2019	CODA [55]	NA	NA	x86-64, MIPS64	clang	O0	None	NA
2019	DIRE [84]	GitHub projects written in C.	164632	x86-64	NA	None	None	Linux
2019	GMN [93]	FFmpeg	NA	NA	clang, gcc	NA	None	Linux
2019	GraphEmb [106]	binutils, ccv, coreutils, curl, ffmpeg, gdb, gsl, libhttpd, openmpi, OpenSSL, postgresql, valgrind	11244 *	x86-64, aarch64	clang-3 (2), clang-4.0, clang-5.0, gcc-3.4, gcc-4 (3), gcc-5.0, icc-17, icc-19	O0, O1, O2, O3	None	Linux
2019	Himalia [36]	399 popular open-source projects (bash, gzip, ntp, openssh, PuTTY, SQLite, etc.)	5828	x86-64	gcc-4, gcc-5, gcc-6	O0, O1, O2, O3, Os	None	Linux
2019	InnerEye [175]	binutils, coreutils, diffutils, findutils, OpenSSL	844 *	x86-64, aarch64	clang-6	O1, O2, O3	None	Linux
2019	SAFE [107]	binutils, ccv, coreutils, curl, gsl, libhttpd, openmpi, OpenSSL, valgrind	5001 *	x86-64, aarch64	clang-3, clang-4 (3), clang-5, clang-6, clang-7, gcc-3 (2), gcc-4, gcc-5, gcc-6	O0, O1, O2, O3	None	Linux
2019	XArchInstrEmb [128]	binutils, coreutils, diffutils, findutils, OpenSSL	804 *	x86-64, aarch64	clang-6	O0, O1, O2, O3	None	Linux
2020	CodeCMR [169]	NA	NA	x86-64, aarch64	clang, gcc	O0, O3	None	NA
2020	DeepBinDiff [49]	coreutils, diffutils, findutils, LSHBOX, Indicators (C++)	2206 *	x86-64	gcc-4	O0, O1, O2, O3	None	Linux
2020	NERO [41]	NA	541 *	x86-64	NA	O1, O2, O3	API name obfuscation	Linux
2020	OrderMatters [168]	NA	NA	x86-64, aarch64	gcc	O0, O1, O2, O3	None	NA
2020	o-glassesX [112]	NA	28074	x86, x86-64	clang-5, gcc-6, icc-19, MVSC 2003, MVSC 2017	O0, O3, Od, Ox	None	NA
2020	Punstrip [117]	coreutils, findutils, moreutils, x11-utils, x11-xserver-utils	2132	x86-64	gcc, clang	Og, O1, O2	None	Linux
2020	TIKNIB [80]	51 GNU packages	243128	x86, x86-64, ARM, aarch64, MIPS, MIPS64, MIPSEB, MIPS64EB	clang-4, clang-5, clang-6, clang-7, gcc-4, gcc-5, gcc-6, gcc-7, gcc-8	O0, O1, O2, O3, Os, LTO, PIE	O-LLVM (BCF, FLA, SUB)	Linux
2021	DIRECT [110]	GitHub projects written in C. (same as DIRE)	164632	x86-64	NA	None	None	Linux
2021	PalmTree [91]	binutils, coreutils, diffutils, findutils	3266	x86-64	clang-8, gcc-9	O0, O1, O2, O3	None	Linux
2021	StackBert [43]	binutils, coreutils, SPEC CPU2017	2076	x86-64, aarch64	clang-13, gcc-11	O0, O1, O2, O3	None	Linux
2021	StateFormer [122]	bash, bc, binutils, bison, busybox, cflow, coreutils, curl, diffutils, dpkg, findutils, gawk, grep, gtypist, gzip, ImageMagick, indent, inetutils, less, GMP, libmicrohttpd, libpng, libtomcrypt, nano, OpenSSL, PuTTY, sed, sg3-utils, SQLite, usbutils, util-linux, wget, zlib	NA	x86, x86-64, ARM, aarch64, MIPS, MIPS64	gcc-8 (2)	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB, IBR, SPL)	Linux

Year	Work	Sources	# Binaries	ISA	Toolchain	Optimizations	Obfuscation	OS
2021	TREX [121]	binutils, coreutils, curl, diffutils, findutils, GMP, ImageMagick, libmicrohttpd, libtomcrypt, OpenSSL, PuTTY, SQLite, Zlib	NA	x86, x86-64, ARM, aarch64, MIPS, MIPS64	gcc-7	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB, IBR, SPL)	Linux
2021	VarBERT [18]	GitHub projects written in C. (same as DIRE)	164632	x86-64	NA	None	None	Linux
2021	XDA [119]	curl, diffutils, GMP, ImageMagick, libmicrohttpd, libtomcrypt, OpenSSL, PuTTY, SQLite, Zlib, SPEC CPU2017, SPEC CPU2006, BAP	> 3121	x86, x86-64	gcc-4, gcc-5, gcc-7, gcc-9, icc-14, MSVC 2008, MSVC 2010, MSVC 2012, MSVC 2013, MSVC 2019	O0, O1, O2, O3, Ox, Od	O-LLVM (BCF, FLA, SUB, IBR, SPL)	Linux Windows
2022	BinMLM [137]	code from Google Code Jam competition, Codeforces, and some malware.	NA	NA	NA	NA	NA	NA
2022	BINSHOT [4]	binutils, busybox, coreutils, diffutils, findutils, ImageMagick, libcurl, libgmp, libtomcrypt, nginx, OpenSSL, PuTTYgen, SQLite, vsftpd, Zlib, SPEC2006, SPEC2017	1400	x86-64	clang-6, gcc-5	O0, O1, O2, O3	None	Linux
2022	Codee [163]	bash, coreutils, libcurl, libgmp, OpenSSL	NA	x86-64, aarch64, MIPS64	clang-3, gcc-5	O0, O1, O2, O3	None	Linux
2022	DeepDI [166]	curl, diffutils, GMP, ImageMagick, libmicrohttpd, SQLite, Zlib, LLVM 11, SPEC CPU2006, SPEC CPU2017, BAP	> 2128	x86, x86-64	gcc-4, gcc-7, MSVC 2008, MSVC 2019	O0, O1, O2, O3, Od, Ox	O-LLVM (BCF, FLA, SUB, IBR, SPL)	Linux, Windows
2022	DIRTY [32]	GitHub projects written in C.	75656	x86-64	gcc-9	O0	None	Linux
2022	HowMLBinSim [105]	binutils, clamAV, coreutils, curl, diffutils, findutils, GMP, ImageMagick, libmicrohttpd, libtomcrypt, nmap, OpenSSL, PuTTY, SQLite, unrar, z3, Zlib	> 5489	x86, x86-64, ARM, aarch64, MIPS, MIPS64	clang-3, clang-5, clang-7, clang-9, gcc-4, gcc-5, gcc-7, gcc-9	O0, O1, O2, O3, Os	None	Linux
2022	JTRANS [155]	ArchLinux and Arch User repositories.	48130	x86-64	clang, gcc	O0, O1, O2, O3, Os	None	Linux
2022	NeuDep [120]	41 open-source projects (bash, bc, binutils, bison, cflow, coreutils, curl, findutils, gawk, OpenSSL, etc.)	NA	x86-64	clang-8, gcc-9	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB, SPL)	Linux
2022	NeurDP [29]	programs randomly generated with cfile	4000	NA	clang-10	O0, O1, O2, O3	None	Linux
2022	Neuro-Debug^2 [12]	programs randomly generated with Csmith	47904	x86-64	clang-13	Og	None	Linux
2022	Sem2vec [154]	binutils, coreutils, diffutils, findutils, libgmp, libtomcrypt, OpenSSL, rapidjson, Zlib	NA	x86-64	clang-4, gcc-7	O0, O2, O3	O-LLVM (BCF, FLA, SUB)	Linux
2022	SymLM [76]	bash, bc, binutils, bison, busybox, cflow, coreutils, curl, diffutils, findutils, ImageMagick, inetutils, less, libgmp10, libmicrohttpd, libpng, libtomcrypt, nano, OpenSSL, PuTTY, sed, sg3-utils, SQLite, wget, Zlib1g	16027	x86, x86-64, ARM, aarch64, MIPS, MIPS64	gcc-7	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB, SPL)	Linux
2022	XBA [81]	Apache HTTPD, curl, glibc, libcrypto, OpenSSL, SQLite	NA	x86-64, aarch64	NA	O0, O1, O2, O3	None	Linux, Windows
2023	VulHawk [103]	coreutils, curl, diffutils, findutils, libmicrohttpd, mtools, OpenSSL, PuTTY, wget, SQLite	3393	x86, x86-64, ARM, aarch64, MIPS, MIPS64	clang-10, gcc-10	O0, O1, O2, O3, Os, Ofast	None	Linux
2023	BinFinder [127]	binutils, coreutils, glibc, GMP, ImageMagick, libcurl, OpenSSL, Zlib, 49 GNU packages from TIKNIB	NA	x86-64, aarch64	clang, gcc	O0, O1, O2, O3	O-LLVM (BCF, FLA, SUB)	Linux
2023	AsmDepictor [82]	utilities and libraries across different Ubuntu Linux distribution versions	3063	x86-64	NA	NA	None	Linux
2023	Callee [174]	GNU Binutils	694	NA	clang-6, clang-12, gcc-7, gcc-9	NA	None	Linux
2023	MAIE [156]	binutils, coreutils, diffutils, findutils, libgpg-error, OpenSSL, etc.	NA	x86-64, aarch64, MIPS64, PPC64	NA	O0, O1, O2, O3	None	Linux
2024	BinBert [13]	binutils, ccv, coreutils, curl, diffutils, ffmpeg, findutils, gdb, GMP, gsl, ImageMagick, inetutils, libhttpd, libtomcrypt, mailutils, nmap, openmpi, OpenSSL, postgresql, pre, PuTTY, SQLite, valgrind, wget, Zlib	222645 for pretraining** 2344 for testing	x86-64	clang-3(2), clang-4, clang-5, clang-6, clang-9, gcc-3, gcc-4(3), gcc-5, gcc-7, gcc-9, icc-21	O0, O1, O2, O3	None	Linux
2024	HermesSim [66]	subset of HowMLBinSim dataset	5489	x86, x86-64, ARM, aarch64, MIPS, MIPS64	clang-3, clang-5, clang-7, clang-9, gcc-4, gcc-5, gcc-7, gcc-9	O0, O1, O2, O3, Os	None	Linux

* size obtained from another paper.

** this dataset is composed of object files.

> the size was partially available.

NA information is not available

Table 3.2. Compiled Dataset.

Observed Gaps in the Raw Dataset

While some efforts attempt to address limitations in raw dataset collection [80], several gaps persist:

- The majority of existing studies focus on Linux and x86 architectures, revealing a need for a more comprehensive dataset that encompasses a broader range of architectures, operating systems, and compiler toolchains;
- Many studies independently recreate raw datasets from scratch, even though the raw dataset content is often independent of the specific tasks. This leads to two main effects:
  - Redundancy in data generation efforts;
  - Challenges in benchmarking and comparing different solutions solving the same task.

3.8.2 Binary Representation

After extracting the raw dataset, researchers must determine the most suitable representation for addressing a specific binary task. This section will focus on the various methods used to represent a binary. There are two main factors that define a binary representation. The first is the granularity of the representation, which involves deciding whether to represent an entire binary, a single function within the binary, a basic block within a function, or a strand inside a basic block. This decision is primarily influenced by the task that the model is designed to address: function-level tasks require a representation of functions, while binary-level tasks require a representation of the entire binary. The second factor is the language used to represent the basic instructions of a binary. This language could be as simple as the raw bytes that compose an assembly instruction or the assembly language itself. Some approaches take a more abstract route, using decompiled code or an intermediate representation of the assembly language. In this section, we will primarily focus on the first factor: how models represent binaries, functions, basic blocks, and strands. The language, along with its preprocessing and tokenization, will be discussed in Section 3.8.3.

Overview of common binary representions. From an high level perspective there are three main ways in which binaries and functions can be represented: a graph structure, a linear representation, or with elements extracted through execution, either concrete or symbolic (see Table 3.3). While basic blocks are typically represented in their linear form; strands have been represented both in their linear form and through execution. Regarding graph structures for programs and functions, ICFG and the CFG, are well-known and widely used and they are the two most prevalent methods. However, there are also more exotics representations, most of which are derived from ICFGs and CFGs, that have been used by reviewed works. It is important to remark that a single work can use multiple representations. This is done for several reasons:

- It may solve multiple tasks that require different levels of granularity (this is

the case of PalmTree and BinBert);

- It may use multiple networks with different representations in parallel. For example DIRE uses both an LSTM on the linear decompiled code and a GNN on its AST representation;
- It may employ multiple pipelined steps to obtain a representation. For instance, OrderMatters first constructs embeddings for basic blocks and then uses these embeddings to represent nodes in the CFG through a GNN;
- It may use one representation for training and another during inference. For example, TREX uses function execution traces during training and a linear representation during testing, while BinBert uses strand and symbolic execution during training and different representations at inference according to the task solved.

In the following sections we will detail all the aforementioned representation at different granularity levels, along with the works that employ them. A visual summary of the different representation used by the analyzed works is shown in Table 3.3.

			Raw Bytes	Assembly	Decompiled	IR
Binary	**Graph**	**IFG**		DeepDI		
		ICFG		DeepBinDiff, SymLM, Codee, BinMLM, BinEye		
		BDG		XBA		
	Execution	**Concrete**	DeepVSA	PalmTree		
	Linear		BinEye, RFBNN, o-glassesX, XDA	BinEye		
Function	**Graph**	**CFG**		Gemini, GMN, GraphEmb, CodeCMR, NFRE, Codee, OrderMatters, PalmTree, MAIE		VulHawk, Punstrip, XFL, BinFinder, NeurDP
		CECFG		Asm2Vec		
		LSFG		VulSeeker		VulHawk
		SOG				HermesSim
		ACSG		NERO		
		DFG		PalmTree		
		AST			DIRE	
	Execution	**Concrete**		TREX, Neuro-Debug2		
		Symbolic		Sem2Vec		Punstrip, XFL
	Linear		Eklavya, adiff, RNND, StackBert	SAFE, Himalia, CODA, TREX, StateFormer, PalmTree, BINSHOT, JTRANS, AsmDepictor, BinBert, Callee, SymLM, NFRE, MAIE	DIRE, VarBert, DIRECT, DIRTY	
Basic Block	**Linear**			InnerEye, XArchInstrEmb, BinBert, CodeCMR, GraphEmb, OrderMatters		VulHawk
Strand	**Execution**	**Concrete**		BinBert		
		Symbolic		BinBert		
	Linear			Zeek, BinBert		

Table 3.3. Categorization of works according to the binary representation used.

Binary-level Representations

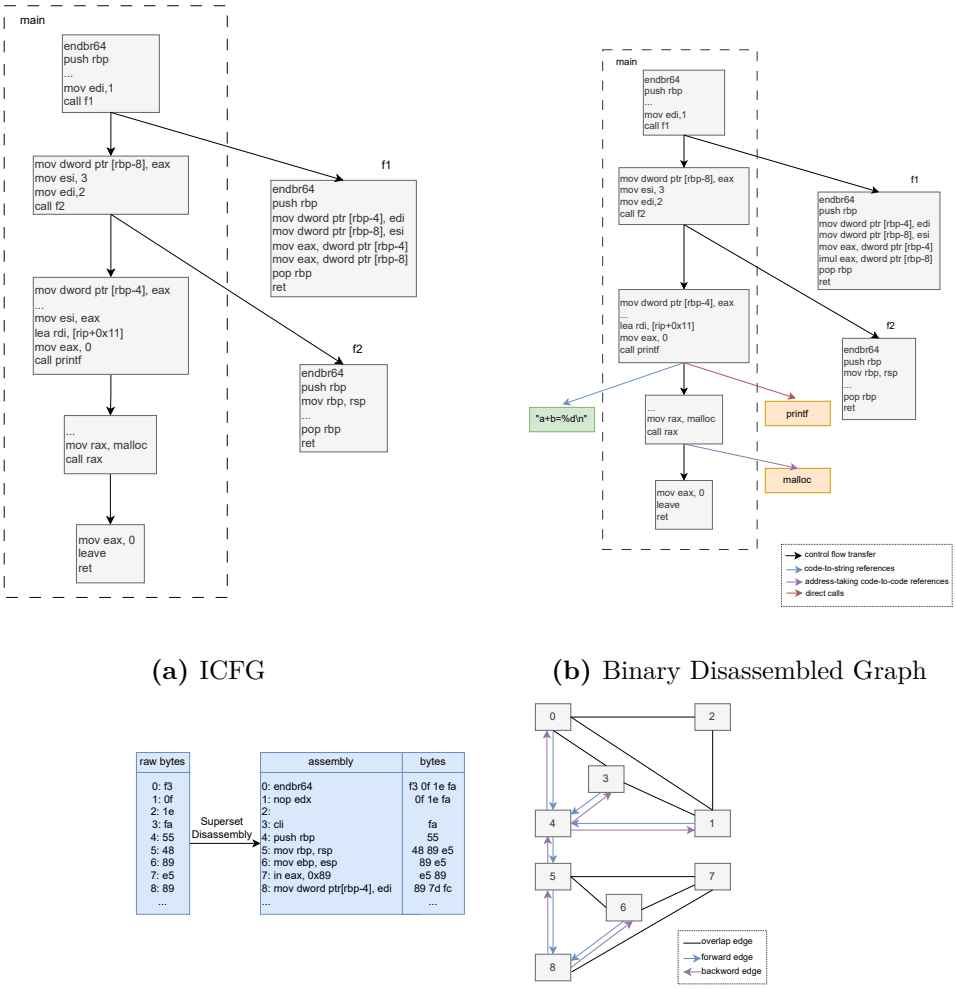
As mentioned before, binary-level representations can be classified into linear, graph, and execution-based representations. Each of these representations will be discussed in the following paragraphs.

Linear Representation. Unceremoniously, a linear representation of a binary is a sequence of information that represents the binary itself. It can either be the entire sequence of bytes composing the binary, in their order in the raw file, or, in its high-level version, the sequence of assembly instructions composing the binary. When we consider very long sequences of bytes or instructions, we must take into account that it is impractical to feed these entirely into a deep learning model. Therefore, all the works we surveyed use some strategy to reduce the size of the information processed in one step. The most common strategy is to use subsequences. For example, RFBNN and XDA solve the function boundary and instruction recovery tasks by using fixed-length subsequences of the entire binary byte sequence under analysis. BinEye solves the code authorship task by using, among others, two linear representations: bytes converted into an image (prior to disassembly) and the sequence of opcodes (after disassembly). Note that we consider, an image to be a linear representation arranged in a grid-like fashion. O-glassesX uses the byte sequence of instructions after disassembly for the compiler provenance task by randomly sampling code fragments of fixed size from binaries.

Graph Representations. A natural way to represent a program is by using a graph to model the relationships between its components. In the literature, the most commonly used representation of this kind is the ICFG. Other notable representations are the *Instruction Flow Graph* (IFG) and the *Binary Disassembled Graph* (see Table 3.3). In Figure 3.6, we provide examples of the three representations discussed above. We will refer to this figure in the following sections to clarify the details of each representation.

One of the main challenges to overcome when using a graph representation is deciding how it should be fed to the model. When the model itself is a Graph Neural Network, there is no need to transform the graph structure, and the only decision is regarding the features to assign to edges and nodes. When the model is not a GNN, the graph structure must be transformed in some way. The most common option is to extract paths from the graph according to some methodology, such as using a series of random walks. These paths are used as sequences that can be fed into most models. We will discuss these techniques in detail in the following.

ICFG. The majority of the studies in our survey utilizes the ICFG to represent a program (see Table 3.3). The ICFG has been explained in Section 3.5. An example of ICFG can be found in Figure 3.6a. Most studies utilizing the ICFG transforms it in a set of sequences by performing random walks on it, the sequences are composed by the assembly instructions encountered during the walks. This is done in DeepBinDiff, Codee, and BinMLM. Other works extract from the ICFG more fine grained information. BinEye utilizes the ICFG to extract the sequence of API calls, while SymLM uses it to extract the calling context of a function. More specifically SymLM extracts a subset of callee and callers of the function they want to name; this is done by taking the top ranked callee and callers according to a frequency based rank. This subset is then represented by using a set of embedding functions computed using a transformer model.



(a) ICFG **(b)** Binary Disassembled Graph

(c) Instruction Flow Graph

Figure 3.6. Examples of different graph representation at binary level.

Binary Disassembled Graph. The *Binary Disassembled Graph* (BDG) is a novel modified ICFG introduced by XBA. A graphical representation is in Figure 3.6b. The BDG is a ICFG in which graph nodes do not only represent basic blocks but also external functions and strings. Besides traditional node relationships, which include control-flow transfer between basic blocks and direct calls, the Binary Disassembled Graph also contains two other types of relationships:

- Address-taking code-to-code references: these relationships happen when a basic block contains instructions that use the address of another function;
- code-to-string references: they connects a basic block with a string node referenced within that block.

Additionally, because XBA ultimately uses a GNN, no further conversion of the graph-based representation is needed. In other words, XBA directly employs a BDG

with bag-of-words features to represent its nodes.

Instruction Flow Graph. The *Instruction Flow Graph* (IFG) is a binary representation used for the disassembly task by DeepDi. Since this representation precedes the disassembly phase, the right assembly instructions composing the binary are not known. Therefore, the IFG is a graph that represent the relationships between all the potential assembly instructions in which a certain sequence of bytes could be disassembled. More specifically the concept of *superset disassembly* [21] is used to disassemble every executable byte offset to determine all the possible instructions (see Figure 3.6c). All such instructions represent the nodes of the *Instruction Flow Graph*. Among these nodes, three type of relationships exists:

- The forward relationship exists between node i and node j if the next instruction of i can be j (for instance i jumps to j or calls j);
- The backward relationship is the same relationship as the forward one but in the opposite direction;
- The overlap relationship happens when instruction i overlaps with j, i.e. the starting address of instruction j is inside instruction i. Intuitively, an overlap edge indicate that a certain pair of instructions is incompatible as the existence of both is impossible.

Each potential instruction is also converted into a feature vector via instruction embedding, preserving its semantic meaning. The feature vectors are propagated through the IFG using a Relational Graph Convolution Network model (R-GCN [132]) to capture neighboring information, and then fed into a classification layer to predict whether the corresponding instructions are valid. All the aforementioned layers are connected and trained in an end-to-end supervised manner.

Execution-based. Concrete execution is employed by DeepVSA to address the Value Set Analysis problem, specifically by training the model on execution traces. Similarly, Palmtree also utilizes concrete execution, as it tests its model on the same task. An execution trace, in this case, is the sequence of machine code or instructions executed at run-time.

Function-level Representations

Analogously to binary-level representations, function-level representations can be classified into graph, linear and execution-based representations.

Graph Representations. Besides, traditional representations (CFG, Data Flow Graph or Abstract Syntax tree), reviewed works also use custom representations, such as Callee-expanded CFG, Labeled Semantic Flow Graph, Semantic Oriented Graph and Augmented Call Site Graph. All such different representations will be outlined in the following paragraphs.

CFG. A definition of the CFG has already been provided in Section 3.5. Similar to our discussion of the ICFG at the binary level, studies in the revised literature either use this graph representation directly or leverage it to extract specific information, such as instruction sequences or manually crafted features. Works using the CFG directly include Gemini, GMN, GraphEmb, CodeCMR, VulHawk, Codee, MAIE and OrderMatters. Additionally, since PalmTree attaches its embedding model under Gemini, it uses the CFG direclty as well. It is worth noting that Codee also performs random walks on the CFG to extract instruction sequences for training a token embedding model, a technique also employed by NFRE. Moreover, OrderMatters uses the CFG to extract basic block pair sequences to pretrain its model on recognizing whether two basic blocks are adjacent (Adjacency Node Prediction). This pretraining task helps the model to better understand the flow of control between basic blocks and instructions. Lastly, BinFinder, Punstrip and XFL use the CFG to extract manual features representing a binary function.

Callee-expanded CFG. A *Callee-expanded CFG* (CECFG), introduced and utilized by Asm2Vec, is a CFG where callee functions are inlined based on a decoupling metric. This decoupling metric determines whether a callee function operates as a utility function[6], indicating its level of independence from the caller. Function that have a low score will be inlined in the caller. However, if the callee function is significantly longer than the caller, it will not be inlined. Asm2Vec extracts instruction sequences from this representation using edge sampling [7] and random walks.

Labeled Semantic Flow Graph. The *Labeled Semantic Flow Graph* (LSFG) have been introduced by VulSeeker. Specifically, a LSFG is a CFG in which edges from the Data Flow Graph (DFG) are added. CFG and DFG edges are distinguished by specific labels associated to the edges. Vulseeker directly utilizes this graph-based representation by feeding it into a GNN, with basic blocks represented by manually crafted features. Additionally, although not explicitly stated in the paper, VulHawk uses a variant of this representation on the IR to extract basic block pair sequences to pretrain its model. This pretraining, known as the Adjacent Block Prediction Task, helps the model understand the data-flow relationships between adjacent blocks.

Semantic Oriented Graph. The *Semantic Oriented Graph* (SOG) is a binary representation proposed by HermesSim to more effectively capture the semantics of binary functions. In this graph, each node represents a token of an instruction (either an opcode or an operand) in the Intermediate Representation (IR), and it incorporates three types of relationships:

- Data relationships exist between nodes that represent tokens of the same instruction. Numbers on the edges indicate the different orders between the opcode and operands. Data relationships also exist between nodes that share a def-use relationship;

[6] A routine used by multiple functions to perform simpler tasks.

[7] They randomly sample edges from the CECFG and for each sampled edge, they concatenate their assembly code to create a sequence.

- Control relationships represent the control flow between instructions;
- Effect relationships capture restrictions on the execution order between instructions.

HermesSim directly use this graph-based representation by using a GNN.

Augmented Call Site Graph. The *Augmented Call Site Graph* (ACSG), introduced by NERO for the function naming problem, is built from the CFG. In this graph, nodes are call sites in the disassembled code,while edges represent the execution order of these call sites. Each call site is actually augmented by reconstructing the register used as arguments and using pointer-aware slicing to retrieve their values. Such values are either concrete or abstract values. Abstract values are an approximation of the real concrete values and fall into the following categories: arguments (ARG), global (GLOBAL), return from calling procedure (RET) or local variable stored onto the stack (STK). NERO directly uses this graph-based representation by using a GNN.

Data Flow Graph. Palmtree uses the Data Flow Graph (DFG) to create instruction pair sequences for pretraining its model on recognizing data relationships. The task involves determining whether a Def-Use relationship exists between instruction pairs.

Abstact Syntax Tree. The Abstract Syntax Tree (AST) built on the decompiled code is used by DIRE for the variable naming task. More specifically DIRE employs an augmented AST in which new type of edges are added: edges from function names to its arguments nodes, edges from terminal nodes to its successor to capture the sequence and edges from variable nodes to a virtual node connecting all occurrences of that variable within the code. DIRE directly uses this graph-based representation by using a GNN.

Linear Representation. The linear representation is the most common representation and consists of the sequence of instructions provided by a disassembler or a decompiler. The disassembly instruction sequence can also be seen as a Linearized CFG, that is the sequence of instructions obtained by ordering basic blocks according to their addresses. The downside of this representation is that blocks and instructions which are not logically related could be place one after the other, thus possibly introducing some noise. The disassembly instruction sequence with raw bytes is used by Eklavya, $\alpha diff$, StackBert and RNND (which actually used smaller snippets rather than entire functions). On the other hand, the disassembly instruction sequence with assembly instructions is utilized by SAFE, Himalia, CODA, TREX, Stateformer, BINSHOT, JTRANS, AsmDepictor, BinBert, and NFRE. Additionally, Palmtree employs this sequence for the function signature recovery task, and although not explicitly stated in the paper, MAIE also uses it to train its instruction embedding model. Furthermore, SymLM leverages this sequence through TREX to generate function embeddings, and Callee extracts assembly snippets from both the callee and the callsite to recover the call graph. The decompiled code

sequence is utilized by VarBERT, DIRTY, and DIRECT (which uses chunks of this representation). Additionally, DIRE employs the decompiled code sequence alongside the AST representation.

Execution-based. Some works in the revised literature utilize function execution information for training their model. Specifically, TREX and Neuro-Debug[2] use concrete execution to extract execution traces, while Sem2Vec uses symbolic constraints for model training. Additionally, Punstrip and XFL leverage symbolic execution to extract manual features for their models.

Basic-block-level Representations

Basic blocks are only represented in their linear form, i.e. by the sequence of instructions within a block. The linear representation is employed by InnerEye and XArchInstrEmb for the basic block similarity task, and by BinBert for the strand recovery task. Furthermore, CodeCMR, GraphEmb, OrderMatters and VulHawk utilize basic blocks to train models that generate embeddings, which are then integrated into other networks to represent functions.

Strand-level Representations

Strand-level representations used in the literature are linear and execution-based.

Linear representation. The linear strand representation is used by BinBert in the pre-training phase and also for the strand similarity and strand execution tasks. Additionally Zeek, uses strands to represent a function.

Execution-based. Strand execution is only used by BinBert. Specifically it uses symbolic execution during the pre-training phase and concrete execution for the strand execution task.

Observed Gaps in the Binary Representation

From the analysis of binary presentations used in the revised literature we idefied two main gaps:

- Proliferation of custom representations: numerous custom representations have been developed across works addressing specific tasks. Unfortunately, they do not compare properly with standard representations. Additionally, the current literature lacks a comprehensive study that compares these various representations across multiple tasks;
- Scarcity of works based on execution information: most commonly used representations are standard ones, such as linear, CFG, and ICFG-based approaches, while only a few works utilize execution-based information. As a result, most studies offer capabilities similar to those of static analysis solutions, leaving the potential of execution-based information in binary representation largely unexplored.

3.8.3 Preprocessing and Tokenization

In this section, we describe how the works we surveyed preprocess and tokenize the basic instructions of the binary representations they use. From the literature, we identified four different ways of interpreting what constitutes a basic instruction in a binary:

- **Raw bytes:** In this case, the basic instruction is simply the raw bytes that compose the binary. Depending on the binary representation used, these may be the bytes of a basic block in the CFG, or the bytes composing a single instruction;
- **Assembly instructions:** Here, the basic instructions are the assembly instructions obtained by disassembling the raw bytes. This is the most common way of representing instructions in the literature, making it the predominant approach (see Table 3.3);
- **Instructions from intermediate representations:** A few works [29, 53, 66, 103, 117, 127] use basic instructions obtained from some intermediate representation, such as Ghidra's PCode [8] or VEX IR;
- **Instructions from decompiled code:** in this case the binary is first decompiled to an high level decompiled code. The instructions of this code are used by the model.

Below, we provide details for each of the categories above, outlining the techniques encountered for preprocessing and tokenizing instructions.

Raw Bytes. In this Paragraph we will first outline the preprocessing and then the tokenization strategies used in the literature for processing raw bytes.

Preprocessing. When using raw bytes for Instruction Set Architectures (ISAs) with variable-length encoding, such as x64, a key decision is whether to pad the byte string to a fixed length. Fixed-length padding is employed in approaches like DeepVSA and o-glassesX. Additionally, instructions can be preprocessed into different formats, such as sequences or other structures. For instance, techniques like BinEye, o-glassesX, and αdiff arrange bytes in an array or matrix format, allowing them to be effectively processed by convolutional neural networks (CNNs). Some approaches, like StackBert, only extract bytes corresponding to opcodes.

Tokenization. In terms of tokenization, most methods treat a byte as the basic token. The only exception is o-glassesX, which uses bits instead due to its use of short sequences of only 16 instructions, where bit-level granularity is feasible.

[8] Ghidra's PCode is an intermediate representation used by the Ghidra software reverse engineering framework.

Assembly. When dealing with assembly instructions, it is essential to apply effective preprocessing and tokenization strategies to manage the vocabulary size. In fact, unlike natural language, which typically has a vocabulary of hundreds of thousands of distinct words, the vocabulary of assembly instructions can grow significantly. This increase is due to the many variations in instructions, especially when considering the numerous possible immediates, offsets, and memory addresses that can associated to a single instruction. Without well-defined preprocessing and tokenization rules, the vocabulary size can become unmanageably large. In the following sections, we will first discuss preprocessing strategies and then explore tokenization methods for handling assembly instructions effectively.

Preprocessing. Assembly instructions typically consist of at least an opcode and often one or more operands. We can categorize the operands into memory references, immediate values, and registers. Consequently, the preprocessing of assembly instructions must take into account specific techniques for the opcode and for each of these categories:

- For opcodes, they are generally left unchanged in most studies. Notable exceptions include CODA, which filters out "nop" instructions, and XBA, which uses an "icall" token to distinguish between indirect and direct calls;
- Regarding immediate values, studies such as Binshot, Codee, DeepBinDiff, Himalia, and XBA replace all constants with a single special token, while others, such as Innereye, MAIE, XArchInstrEmb, and NFRE, differentiate between negative and positive values. NFRE, in particular, also employs a special token for the zero value. Additionally, works like BinBert, GraphEmb, JTrans, Neuro-Debug[2], PalmTree, and SAFE retain immediate values within a specified threshold, substituting all others with a special token. This threshold-based approach is motivated by the likelihood that small constant values carry informative content (e.g., comparisons with small constants in branches and loops, or PC/stack-relative displacements identifying variables in memory). Sem2Vec, on the other hand, applies logarithmic normalization to map values to a smaller set. TREX and StateFormer replace all values with special tokens within the assembly sequence and transfer them to another sequence, which is processed by a separate network;
- Registers are typically left unchanged. However, some works that adopt more aggressive preprocessing apply normalization rules to further reduce vocabulary size. For instance, Himalia replaces general-purpose and control registers with two distinct special tokens. In contrast, Binshot, Codee, DeepBinDiff, and XBA replace registers with special tokens that retain only information about the size of the register used. More specifically, Binshot uses different tokens for stack, base, and instruction registers;
- Memory references in assembly code can be classified as either direct or indirect. In direct addressing, the memory address of the operand is explicitly specified in the instruction. In contrast, indirect addressing occurs when the memory address is determined by a combination of registers or memory locations, often

using base registers, index registers, and displacement values. Some works do not consider these distinctions and instead replace all memory addresses with a single special token, as in the cases of XBA, DeepBinDiff, Codee, and Neuro-Debug^2. More specifically, XBA differentiates the sizes of accessed memory using different tokens. Other works, such as GraphEmb and SAFE, only replace direct addressing with a special token. Conversely, Himalia uses three different tokens to distinguish between direct memory access, addressing modes with base and index registers, and those with base, index, and displacement. Additionally, each memory reference can represent the address of a function, a jump target, a string reference, a static variable, or other data types. In this regard, IDA uses prefixes to help the analyst differentiate these types of memory addresses. Some works, such as Calle and MAIE, leverage this information by retaining these prefixes during preprocessing. Specifically, while MAIE only retains the prefixes, Calle also keeps the memory reference by applying a *modulo N* normalization. For memory references representing jump targets, BinBert, Binshot, Innereye, and NFRE replace them with a single special token, while Himalia differentiates between far and near addresses. JTRANS replaces jump targets with a special token that tracks the position of the targeted instruction within the sequence. Regarding call memory addresses, works like Innereye, JTrans, MAIE, and XArchInstrEmb replace them with a unique special token, while Asm2Vec, BinBert, Binshot, NFRE, and XBA retain libc function names. These function names are typically preserved in stripped binaries due to dynamic linking. Regarding strings, similarly to call addresses, some works (Binshot, Innereye, JTrans, MAIE, PalmTree and XArchInstrEmb) use a special token, while others substitute the address with the string itself (DeepBinDiff, Neuro-Debug^2). This is because strings are likely to carry informative high-level information. Finally, Binshot and MAIE use a special token also to identify static variables.

Additionally, MAIE is the only work that also define ISA-specific rulels. For instance, the ARM architecture use curly braces to perform the same operation on multiple registers (e.g. PUSH {R4, R5, R6}). In this case, MAIE expand the single instruction into multiple ones. Another example in both ARM and MIPS architectures, is represented by registers which are aliases of others (e.g. SP which is an alias of R13 and R29 in the two architectures respectively). In this case, MAIE removes all the aliases.

Moreover, some works focus exclusively on opcode (e.g., BinEye, BinMLM, GMN) or API call (e.g., BinEye) sequences.

Finally, it is worth mentioning that AsmDepictor is the only work that demonstrates, through an ablation study, that the best performance is achieved without any form of normalization.

Tokenization. Regarding tokenization, we have identified four strategies:

- Istruction based - the entire instruction is a considered as a token. This approach is adopted by Binshot, InnerEye, MAIE, SAFE, XArchInstrEmb and XBA;

- Opcode and operand based - opcode and operands are sperate tokens. This is the case of Asm2Vec, Codee and GraphEmb;
- Puctuation based - the tokenization is performed on puctuations, meaning that a single operand can be split into multiple tokens. This approach is used by CODA, JTRANS, Neuro-Debug[2], PalmTree, StateFormer, SymLM and TREX;
- Automatic tokenization - this tokenization scheme uses algorithms such as WordPiece or Byte Pair Encoding (BPE) and represents the standard in the NLP community. This algorithms are used by AsmDepictor and BinBert.

Decompiled Code. In the paragraphs below we will first illustrate the preprocessing and then the tokenization strategies used for the decompiled code.

Preprocessing. Decompiled code is primarily used in studies focused on recovering variable names. One preprocessing technique identified is replacing variable names with placeholders, a method employed by DIRE.

Tokenization. In terms of tokenization techniques, DIRE and DIRECT use SentencePiece, whereas DIRTY and VarBert use BPE.

Intermediate Representation. In the text below we will first illustrate the preprocessing strategies and then the tokenization ones for the Intermediate Representation.

Preprocessing. Preprocessing rules for Intermediate Representation (IR) have been established by works such as HermesSim and VulHawk, which utilize different IRs—specifically, Ghidra Pcode and IDA Microcode. Consequently, their preprocessing rules may vary, though a common approach can still be identified. Both works employ a frequency-based method to include tokens in the vocabulary, with tokens outside the vocabulary being replaced by root tokens that capture their basic semantics. Furthermore, VulHawk normalizes addresses by substituting them with a designated token.

Tokenization. Regarding tokenization, VulHawk explicitly splits instructions into an opcode and three operands, whereas HermesSim does not provide specific details about its tokenization strategy.

Observed Gaps in the Preprocessing and Tokenization

Although some efforts have been made to study the effects of tokenization and preprocessing on assembly code [13, 82], many studies in the literature tend to redefine their own rules, often without adequate comparison to prior works.

3.8.4 Feature Extraction

Feature extraction involves converting a preprocessed binary representation into a vector of real numbers. In the reviewed literature, this process is achieved using both precomputed features and features trained together with the network. Specifically, precomputed features include manually crafted features extracted from the binary representation and features derived using unsupervised embedding methods. These precomputed embeddings are then kept fixed (frozen) and used as input for a subsequent deep neural network. In contrast, features trained together with the network typically involve a downstream embedding layer, often implemented as a learnable matrix, or another underlying network that is trained end-to-end along with the final task.

Manually Crafted Features. Similar to the approach presented in [80], manual features can be categorized into presemantic and semantic features. Presemantic features include properties that can be derived directly from the disassembled code without the need for additional analysis. In contrast, semantic features necessitate more advanced analysis techniques, such as symbolic or dynamic execution, to be extracted. Both presemantic and semantic features can be numerical or cathegorical. Additionally, some works use basic block-level features, while others use features at function level. For example, Gemini and Vulseeker employ numerical presemantic features, which are either derived from the syntax (e.g., the count of call and arithmetic instructions) or from the structure of the binary representation (e.g., the number of outgoing edges in a basic block). Similarly, TIKNIB and BinFinder utilize numerical presemantic features, but at the function level (e.g. number of callers and callees). In addition to numerical features, BinFinder also incorporates categorical features, such as the list of libc calls or the list of unique VEX instructions. XBA employs a bag-of-words representation of instructions within a function, which is classified as a categorical presemantic function-level feature. Additionally, Sem2Vec enhances a precomputed function embedding with two additional manual features: a bit indicating whether the call stack is empty (a semantic numerical feature) and a one-hot encoding of external function calls (a categorical presemantic feature). Moreover, Zeek uses an hash value of the strands composing a function. Since strand computation, requires a semantic analysis, this feature is considered semantic and cathegorical. Finally, Punstrip and XFL use both presemantic and semantic features. Specifically, they both use symbolic execution to extract semantic features.

Pre-computed Unsupervised Embeddings. To precompute features, many studies utilize the word2vec model. This approach is employed by works such as Eklavya, InnereEye, GraphEmb, SAFE, XArchInstrEmb, Bineye, DeepBinDiff, NFRE, and Codee. However, more recent studies, including OrderMatters, PalmTree, Sem2Vec, SymLM, and VulHawk, adopt a more sophisticated approach based on pre-trained transformer architectures. For example, OrderMatters and VulHawk pre-train a transformer network on a specific pre-training task and then use it to generate basic block embeddings, which are subsequently fed into a Graph Neural Network.

Features Trained with the Network. As previously mentioned, some works train features alongside the network by employing an embedding layer in the form of a learnable matrix at the base of another network, such as an RNN. Examples of this approach include RFBNN, HIMALIA, DIRE, and BinMLM. Another approach involves using CNNs, where the convolutional layers are trained together with a feed-forward network to extract features. This method is utilized by $\alpha diff$, BinEye, and o-glassesX. In addition, some works, like DeepVSA and CodeCMR, adopt a hierarchical approach, where they use another network to learn the embeddings. Furthermore, many studies automatically learn features using transformer architectures, as seen in TREX, VarBert, StateFormer, DIRECT, XDA, StackBert, Binshot, JTRANS, DIRTY, AsmDepictor, and BinBert. Finally, other works that use features trained alongside the network include GMN, Asm2Vec, NERO, SymLM, DeepDI, NeurDP, and HermesSim.

Observed Gaps in the Feature Extraction

Although TIKNIB [80] has recently claimed that manually crafted features can achieve good performances in the binary code similarity task, our survey reveals a growing trend toward eliminating the use of manual features. Instead, recent approaches increasingly focus on automatically learned features, trained directly with the network. Thus, further research is needed to explore the full potential of automatically learned features in binary code analysis tasks, investigate their generalizability across different tasks and datasets, and determine whether they can consistently outperform manually crafted features in various real-world scenarios.

3.9 Deep Learning Models

Regarding deep learning models employed by works in the revised literature we observe that, while some works opt to use standard unmodified networks, the majority of works apply custom modification to standard architectures.

3.9.1 Standard Networks

The analyzed works in the literature based on standard networks manly rely on six different neural network types: RNN, Feed Forward Networks, CNNs, autoencoders, Transformers and graph-based networks. Specifically, RFBNN, Eklavya, RNND, InnerEye, SAFE, HIMALIA, DeepVSA and NFRE use standard RNN-based architecture. Asm2Vec uses the PV-DM model [86], $\alpha diff$ uses a CNN, Zeek and BinFinder rely on Feed Forward Network while XFL uses an autoencoder followed by a multilabel classifier. Among works using unmodified transformer networks we can find XDA, StackBert, PalmTree, Neuro-Debug^2 and BinBert. Finally, regarding standard graph-based networks, GEMINI and GraphEmb use the structure2vec network [40], XBA uses a Graph Convolutional Network (GCN) [83], HermesSim uses a Gated Graph Neural Network (GGNN) [94] and DeepBinDiff uses the TADW model [162].

3.9.2 Custom Networks

Most studies in the revised literature focus on modifying existing networks. The sole exception is GMN, which introduces a novel graph-based network. Unlike traditional methods that compute embeddings of graph input pairs separately before determining their similarity, GMN computes a similarity score directly on the input pairs in a joint manner.

We have identified three key types of customizations implemented in the reviewed works:

- Modifications made to the internal structure of existing networks;
- Custom combinations of encoder-decoder architectures;
- Integration of multiple networks in a unique configuration.

In the following sections, we will discuss each of these approaches in detail.

Internal Modifications. Works under this cathegory, mainly operate modifications to the Transformer architecture (JTRANS, AsmDepictor, DIRTY, StateFormer, TREX, VarBert, DIRECT). However, we also identified two works, Vulseeker and o-glassesX, that operate modifications to the structure2vec network and to CNN.

More precisely, Vulseeker slightly changes the node embedding computation in the structure2vec network so as to handle two different sets of neighbouring nodes (i.e. nodes connected by control flow or data flow relationships), while o-glassesX adds positional encoding and attention mechanism to a CNN.

Regarding transformer modification, JTRANS implements in the transformer network a parameter sharing mechanism between jump instructions and their targeted instructions by using the embedding of the former as the positional embedding of the latter. The rational behind this choice is to connect source and target jump instructions, which are usually distant in the flat CFG representation. AsmDepictor applies 3 main modifications to the standard transformer encoder-decoder network. First of all, it reduces the number of stacked layers, secondly it repeats the position embeddings at the beginning of each layer and lastly it modifies the softmax function to better balance attention values. Another example is represented by DIRTY which uses a modified transformer encoder-decoder network to solve the variable naming problem. The first modification lies in the encoder part, in which they introduce a pooling mechanism to produce a unique representation of variables appearing in multiple decompiled code locations. Additionally, at the decoder side, they train another transformer-based network, referred to as Data Layout Encoder, which is responsible of producing a softmax used to reduce the likelyhood of less probable variable types. StateFormer and TREX use a modified transformer which laverages separate networks for handling concrete numerical values in the assembly sequence, specifically StateFormer uses a Neural Arithmetic Unit (NAU) [104], while TREX uses a bi-LSTM network. Other works based on slightly modified transformer networks are VarBert and DIRECT which solve the variable naming problem. More precisely, VarBert uses a variation of the masked language modeling task, named Constrained Language Modeling to recover variable names. Moreover, since the number of tokens composing a variable name is unkwown at inference time, they

introduce a heuristic approach to estimate that number. On the other hand, DIRECT addresses the challenge of variables consisting of an arbitrary number of tokens by employing a dedicated transformer decoder for each variable. Additionally, since a variable name must be the same for all of its occurences, DIRECT implements a customize beam search algorithm to choose the most probable name.

Custom Encoder-Decoder combinations. Depending on the task, some studies need to handle multiple types of sequences. For example, the function naming task involves both assembly sequences and natural language name tokens. In these cases, encoder-decoder networks are commonly employed, with some works utilizing custom combinations of encoders and decoders. For instance, in order to solve the variable name recovery problem, DIRE uses two encoders: a Bi-LSTM encoder on the flat decompiled code sequence (lexical encoder) and a GGNN encoder [94] on a modified AST of the decompiled code (structural encoder). The decoder of DIRE is an attention-based LSTM which outputs identifier names. Another example is CODA which uses a N-ary Tree-LSTM encoder and a tree-LSTM decoder [141]. NERO and NeurDP employ a GNN based encoder and an LSTM decoder. More specifically, the GNN used by NERO is a GCN [83] while the one used by NeurDP is a GGNN [94].

Combination of Multiple Networks. Some works use a combination of existing networks to either process different input types separetely or to aggregate embeddings produced at different stages.

For instance, BinEye uses multiple CNNs to process 3 different binary represenations (binary converted to an image, sequence of opcodes and API calls).

OrderMatters use three main components to process a binary function: a semantic aware component based on a pretrained transformer model which is used to produce basic blocks embeddings, a structural aware component that uses a Message Passing Neural Network [58] to compute the graph embedding starting from node embeddings and an order aware component which produces embeddings of the adjacency matrix of CFGs by using a CNN called Resnet [67]. The final function embedding is computed by concatenating the structural aware embedding with order aware embedding and by using a feed forward network on it.

Similarly to OrderMatters, CodeCMR uses the HBMP [142] model to produce node embeddings and then use a GGNN [94] to produce a representation for the whole graph. Additionally, CodeCMR represents integers and strings separately using two LSTM based networks. The final embedding is obtained by concat and batch normalization of the three embeddings.

Other examples are Vulhawk and DeepDI. VulHawk uses a GCN [83] on the CFG with initial node embeddings produced as mean pooling of the second last layer of hidden states of a transformer previously pretrained on some specific tasks. DeepDI uses a RNN to create instruction embeddings and then a Relational-Graph Convolutional network (R-GCN) [132]. Since this network is used to solve the disassembly task, on top of each node it has a fully connected layer with a sigmoid function to output the likelihood for each instruction of beeing a valid one.

Sem2Vec is based on a three steps approach. In fact, it first uses a transformer pre-trained on symbolic execution contraints and fine-tuned with a siamese network

by matching symbolic constraints that come from the same line of source code. Thein it produces embeddings for tracelets by choosing their K longest symbolic constraints and creating a unique embedding using the HBMP network [142]. Then they concatenate the resulting embedding with a one hot encoding vector representing external function calls inside the tracelet and with a bit representing wether the next block of a tracelets belongs to the same function or not. Finally, in order to compute the embedding for a function, they use a GGNN [94] on its tracelets.

SymLM uses TREX to produce token embeddings and then a pooling scheme to create function embedding. Additionally, in order to make the function embedding aware of the binary context, SymLM creates an embedding lookup matrix for external functions which is trained together with the network. For internal functions, it uses TREX to produce their embeddings. The final embedding is obtained by concatenating all these embeddings.

Observed Gaps in the Deep Learning Models

Most of the revised studies utilize custom networks, often resulting in complex architectures without sufficient comparison to baseline approaches. Therefore, further research is needed to determine whether these intricate architectures are really necessary for specific tasks or if there exists a specific architecture that can effectively represent binary code and address multiple tasks with good performance.

3.10 Pre-training Tasks

As discussed in Section 2.2.2, the pre-training and fine-tuning paradigm has become the standard approach for NLP tasks. This paradigm involves first training a model on a large-scale unsupervised task, and then the pre-trained model is fine-tuned on a specific downstream task. One of the key advantages of this approach is that downstream tasks typically require manually annotated data, which can be difficult to obtain in large quantities. Pre-trained models have been shown to significantly reduce both the amount of labeled data needed for effective performance on downstream tasks and the number of different task-specific architectures. [44].

Similar considerations apply in the context of binary code analysis. However, unlike NLP, in binary analysis, data for some downstream tasks can be generated programmatically without requiring human annotation. For example, a dataset for toolchain provenance tasks can be created by compiling source code and storing toolchain-related information. Nevertheless, programmatically generated data can sometimes be inaccurate (e.g. function names may be imprecise depending on the programmer) or incorrect (e.g. in function-level binary-to-source code matching tasks, the source line information extracted from debug data can be erroneous [45]).

For these reasons, the revised works based on the transformer architecture have used some pre-training tasks. Specifically, besides small variations of the traditional Masked Language Modeling (MLM) task, these works also use different and custom pretraining objectives.

We categorized them into 5 clusters: CFG related tasks, compilation tasks, execution related tasks, data-flow related tasks and semantics tasks. Each of these cathegories will be analyzed in detail in the following paragraphs.

CFG Related Tasks. These tasks require the model to predict properties that can be derived from the CFG. For instance OrderMatters uses two tasks: the Adjacency Node Prediction (ANP) and Block Inside Graph (BIG). The ANP task consists in predicting wether two blocks are adjacent or not, wheter the BIG task consists in detecting wether two blocks exists in the same graph.

JTRANS implements the Jump Target Prediction (JTP) task in which, given a randomly selected jump source token, the model is required to predict its corresponding target token.

Finally, PalmTree uses the Context Window Prediction (CWP) task which consists in making the network recognise if a pair of instructions is taken from the same context window or not.

Compilation Tasks. These tasks force the model to recognise the configuration settings of a specific binary code. One of the works using this kind of task is OrderMatters, which uses the Graph Classification (GC) task to classify blocks in different platform, architectures and optimization.

Another example is Neuro-Debug[2] that relies on the task named Asm/ Source Mapping Prediction task, where the objective is to have the network predict whether a given sequence of assembly instructions is correctly mapped to a certain source line. This task is considered a compilation related task since the mapping information is written at compile time, and thus it represents the way in which the compiler associate a certain assembly code snipppet to the source code lines.

Execution Related Tasks These tasks rely on information extracted during the binary code execution, so as to make the network aware of the effect that specific instructions have at runtime. For instance StateFormer uses the Generative State Modeling (GSM) task which consists in predicting masked micro trace concrete values and in detecting wether a particular instruction is executed in a trace or not. Similarly, TREX is based on the task of recovering micro trace values. Finally, BinBert uses a symbolic execution task named, Strand-Symbolic Mapping (SSM), in which the goal is to predict whether the symbolic expression matches the strand.

Data-flow Related Tasks. These tasks are based on data-flow information and are used by PalmTree and VulHawk. Specifically, PalmTree uses the Def-Use Prediction (DUP) task in which the network has to recognize if there is a data dependency between instructions. Similarly, VulHawk relies on the Adjacent Block Prediction (ABP) where, given a basic block A that contains a variable definition and a subsequent basic block B in which that variable is used, the task is to label the order of A-B as positive and the order of B-A as negative.

Semantics Tasks. These tasks rely on semantic aspects of instructions. The only work we observed that is based on this task is VulHawk. In particular, VulHawk utilizes the Root Operand Prediction (ROP) task which consists of associating the operand to its semantic root type. The authors divided the operands into 16 types.

Observed Gaps in the Pretraining Tasks

Regarding works relying on pre-training, we observe two main gaps:

- Development of new pre-training tasks: several studies introduce new pre-training tasks without adequately comparing their effectiveness to the standard masked language modeling (MLM) task;
- Evaluation on a single downstream task: with the exception of BinBert and PalmTree, most studies in the literature apply the pre-training and fine-tuning paradigm to solve only a single task. However, one of the primary benefits of this approach is the ability to define a unified architecture capable of addressing multiple tasks.

3.11 Conclusion

In this chapter, we present a systematization of 54 research papers that explore the application of deep learning techniques to binary analysis tasks. Despite being a fundamental and promising area of research, few efforts have been made to systematically organize the diverse solutions proposed in this field. Existing surveys either do not focus specifically on deep learning approaches [161], or they concentrate on single specific tasks [105]. In this work, we advance the field by offering a comprehensive review that spans nine years of research, up to 2024. We identify a common deep learning pipeline and provide an in-depth analysis of each step, highlighting key trends across the various approaches as well as gaps that need further investigation.

Chapter 4

Debugging Debug Information with Neural Networks

4.1 Introduction

Software running in production is highly optimized to maximize its performance according to several metrics. For compiled languages, binaries are the output of a compilation process where several optimization techniques are applied. While these optimizations are critical for the performance of the produced artifacts, they may expose unwanted behaviors observable only in the optimized case (e.g., race conditions [73], use-after-free [48], and heisenbugs [165]).

Therefore, debugging the exact version of the binary running in production is key to triaging specific problems that are otherwise impossible to reproduce. Hence, it is crucial to have a complete and reliable debugging process for optimized binaries [68]. In order to debug, users need a compiler and a debugger. The compiler, when instructed to do so, produces a binary that is composed of the binary code and additional debug information, which for UNIX-like systems is usually encoded using DWARF [51]. This information is then used by the debugger during the debugging phase. It is worth noting that while debuggers are the main users of debug info, also tools consume them too, e.g. profilers.

Preserving the correctness of debug information while optimization passes are applied is an extremely complex task. To address this challenge, compilers introduced new optimization levels (e.g., -*Og* in GCC and clang) specifically targeted at providing a reasonable tradeoff between the debugging process and optimizations applied. As a matter of fact, *Og* is described as the optimization level for the standard edit-debug lifecycle in the GCC documentation.

However, it has been recently shown [45, 92] that modern optimizing toolchains[1] often provide an unsatisfactory debug experience even when using optimization levels specifically designed for debugging. This happens when the compiler generates wrong debug information or when the debugger does not handle correctly the debug information produced. Therefore, it is important to examine debug traces looking for problems that can be generated by a bug in the toolchain.

[1] An optimizing toolchain is the union of a compiler capable of optimizing code and a debugger.

Recent works [45, 92] used a differential testing approach where optimized and unoptimized binaries are compiled from the same source. The debug traces of the two binaries are then compared using manually defined invariants. These invariants look for the inconsistency of some information to identify suspect cases. This approach can only find bugs that impact the controlled information as predicted by the invariants' designers.

Therefore, it would be of extreme interest to build a technique for the automatic detection of incorrect debug traces without the use of manually defined rules. A deterministic solution to this problem would entail defining a formal model of the optimization passes of the compiler. This is far from trivial even limiting the scope to a single optimization pass.

In this paper, we take a data-driven approach, in which we use a large dataset to learn a statistical model of correct debug traces. The advantage of this approach lies in its black-box nature: it does not need knowledge nor makes any assumption about the internal structure of the compiler and debugger. We are interested in unsupervised techniques because a labelled dataset of correct/bugged debug traces is not available, and it cannot be generated automatically.

Specifically, we use neural networks following an anomaly detection approach. We train several models in an unsupervised way on a dataset of collected debug traces. Our hypothesis is that the networks may learn the relationships contained in correct debug traces. This has been inspired by works using a similar approach to find bugs in source code [6, 89].

We test our models on two novel datasets. A synthetic dataset of bugged debug traces and another one obtained from real bugs from the LLVM repository. Our experiments confirm that our models discriminate between bogus and correct traces. Finally, we test our models in a *live analysis* to find new bugs in the widely used LLVM toolchain (composed by the clang compiler and lldb debugger).

4.1.1 Motivating example

Snippet 4.1 shows a bug of LLVM found with our solution. The bug is present in the latest LLVM when compiling with optimization *-Og*. When stepping over instructions, the backtrace information of the lldb debugger wrongly shows that line 3 is inside the main function (this is probably the effect of inlining). This kind of bug could mislead a developer that sees a return instruction executed right at the beginning of the `main`, inferring that the rest of the instructions in that function will not be executed.

```
1  volatile int a,  g_5108; int b;
2  short c(){
3   return g_5108;
4  }
5  int main () {
6   c();
7   b = 0;
8   for (;
9   b < 4;
10  b++) a ;
11 }
```

Snippet 4.1. Clang bug 51511, wrong backtrace information at line 4.

4.1.2 Contributions

We provide the following contributions:

- We are the first to consider the problem of detecting bugs in debug information using DNNs. We use transformer architectures, trained with novel unsupervised tasks, to create (i) a network that is able to identify a wrong stepping behavior on a debug trace (*SLNet*) and (ii) a second one that is able to identify an incorrect mapping between assembly instructions and source code (*MapNet*).
- We release three new datasets [2]: a large unlabelled dataset constituted by debug traces, a dataset with debug traces containing synthetic bugs, and a manually labelled dataset containing real bugs.
- We conduct an experimental evaluation of the proposed architecture on the aforementioned dataset.
- We use our best-performing networks in a novel fuzzing system. We reported 12 bugs found in the LLVM toolchain: 2 of these bugs have been confirmed and 10 are pending analysis by the LLVM developers.

4.2 Related Work

4.2.1 Compiler Toolchains Testing

The problem of testing compiler toolchains can be divided into three main branches: compiler, debugger and debug information testing. While compiler testing has been widely investigated, little attention has been paid to identifying bugs in debugger software and debug information of optimized binaries.

Compiler Testing Compilers are widely used and complex software; this complexity makes them prone to bugs. Since compilers are used to build other production software, these bugs could result in unwanted behaviors, or worse in security-related problems [72], in different applications. For these reasons, there is a heap of works focusing on finding compiler bugs [30, 35, 37, 97, 134] using compiler testing.

Compiler testing techniques [31] are mainly based on fuzzing: compilers are fed with random programs either generated from scratch [14, 164] or obtained by mutating existing ones [87, 88]. The generation from scratch uses rules defined for the grammar of the specific language, mutation-based techniques apply mutations to programs generated with the first approach. A compiler bug can either generate a crash or create a program with unwanted behavior; the first case can be easily detected. The second case is usually detected using differential testing [20, 72], which is based on comparing the results or the behavior of programs obtained from different compilers. Another approach is metamorphic testing [33], which is based on the idea of transforming the input while detecting unexpected behavioral changes.

Some novel approaches follow the *Big Code trend* [6] and are based on NLP techniques. In particular, [39] and [101] use LSTM architectures for generating and

[2] https://github.com/FiorellaArtuso/NeuroDebug-2_Dataset

mutating test cases. We remark that they only produce test cases and do not use neural networks to detect bugs.

Debugger Testing Regarding debugger testing, [90] tests the correctness of javascript debuggers using a differential approach and assuming that different debugger implementations should exhibit the same behavior. Operatively, they execute the same debugging actions in parallel and they compare the corresponding results: every diverging behavior identifies a possible bug. Notice that this approach is not able to catch debug information bugs, if a debug information is wrong it will result in the same wrong behavior on different debuggers.

[143] proposes a metamorphic approach for testing the debugger in the Chromium browser. They transform both the input program and the debugging actions and they detect whether this transformation causes unexpected changes.

Debug Information Testing There are only two works that identify debug information errors in optimized binaries [45, 92] and none of them used neural approaches. Both works are based on differential testing and tackle the problem using likely invariants. In this approach, a source code program is compiled with and without optimization obtaining an optimized binary and an unoptimized one. Both binaries are executed using a debugger and recording the execution traces resulting in an optimized trace and an unoptimized one. A likely invariant compares these traces and triggers when a specific inconsistency is detected. This inconsistency could be caused by a bug in the debug process. An example is the Line Invariant of [45]; this invariant takes the unoptimized trace and the optimized one and triggers when there is a line appearing only in the latter. This could be dead code wrongly shown as executed. We stress the invariant is likely, thus false positives could be possible; this is true for the other invariants present in [45] as well. To the best of our knowledge, no technique exists to automatically create invariants for debug information.

4.2.2 Neural Bug Finding

Several papers [64] used Deep Learning to find bugs in source code across several languages. We remark that these works focus on finding a bug inside the code and they do not detect wrong debug information.

An example is the so-called variable misuse bug that mainly occurs when the developer copies pieces of code from one place to another and forgets to adapt the used variables to the new location. To solve this problem, [6] represents a program using a semantic enriched Abstract Syntax Tree (AST) and uses a Graph Neural Network (GNN) on top of it to predict which variable should be used at a specific location. Another work is [148], which uses a two-pointer attention-based LSTM to jointly predict the bug location and the repairing variable.

Some other works focus on fixing errors arising at compile time (e.g. missing brackets). The problem with this type of errors is that compilers do not always show the correct error message or the correct bug location, thus misleading the programmer. [64] solves the problem by using an attention-based sequence-to-sequence network

that takes as input the sequence of tokens representing the program and produces as output both the bug location and the correct fixed version of the bugged line.

Recent works [24,47] use the concept of pretraining and finetuning of transformer-based architectures to generate fixes for different kinds of bugs.

Another work is [89], which proposes a Language Modeling based fuzzer for the javascript engine trained on regression javascript tests. In particular, the authors splits the AST of the regression test cases into subtrees and, by performing a preorder traversal of the tree, obtain sequences of such subtrees. The goal of their modified language model is to make the network predict which subtree comes next. They generate new test cases by mutating existing regression test cases by using the obtained language model. With this approach, the authors find lots of bugs and 3 CVE.

4.3 Debug Trace, Problem Definition and Overview

Given a source code c, an optimizing compiler generates an optimized program *opt*. From this program, a debugger produces a debug trace $T(opt) : [s_0, s_1, \ldots, s_n]$ which is an ordered list of elements, each representing a *step* over a machine instruction. $T(opt)$ is built through instruction by instruction step execution: we set a breakpoint on the program entry point and repeatedly step over assembly instructions until *opt* exits. At each step $s \in T(opt)$, we collect the source line $line(s)$ that the debugger shows in program c when executing s. This is the high-level source line of code in c that the debugger believes to be executed with step s. Moreover, we also collect the assembly instruction $asm(s)$ that the CPU executes in s. A c program is constituted by several functions $\{f_1, \ldots, f_m\}$; we divide an execution trace T into several traces $T_{f_1}, \ldots, T_{f_m}$, where T_{f_j} represents the execution of function f_j in trace T.

4.3.1 Preliminary Definitions

Before moving on to the problem definition, let us provide some necessary elements.

Sequence of Source Lines Each function trace T_f defines a sequence of executed source code lines. Formally, $L(T_f) : [l_1, l_2, \ldots, l_k]$ is the *sequence of source code lines* present in trace T_f in the order of their appearance (a line l_j appears when a step $s_j \in T_f$ has $l_j = line(s_j)$); we remove loops by deleting consecutive sequences of repeated lines.

Mapping Assembly and Source Code For each line l in c, we have associated a sequence of steps, each of which is a step over one of the assembly instructions used to execute line l. Thus, from T_f we extract a set $A(T_f)$ of *mapping pairs* $< [a_0, \ldots, a_m], l >$ where $[a_0, \ldots, a_m]$ is the sequence of all assembly instructions that the debugger maps to line l in trace T_f.

Optimization Levels Modern compilers provide several optimization levels. In this paper, we focus on *Og* that is a level created to provide a good debug experience, i.e. facilitate the debugging process by embedding accurate and complete debugging

information in the compiled software, while creating optimized binary code. Og is supported by the two main C compilers available today: GCC and clang. We consider Og as it was shown to be the optimization level more prone to bugs [45].

4.3.2 Problem Definition

A pair compiler/debugger can be seen as a toolchain function F that maps each source code c to a debug trace $T = F(c)$. The set $Traces$ of all possible traces generated by F for all valid programs [3] can be partitioned into the set of correct traces $NoBug$ and the one of bugged traces Bug. We now define the *Correct Debug Detection* (CDD) Problem:

Correct Debug Detection Problem: Given a trace $T \in Traces$ a CDD gives as output 0 if T is a correct debug trace ($T \in NoBug$) and 1 if T is a bugged debug trace ($T \in Bug$).

It is worth noticing that a perfect toolchain F should never generate a bugged trace. We remark that our final goal is to find bugs in the toolchain, and so to find instances of programs that generate wrong debug traces under F. We are not concerned about the correctness of programs use to test F.

4.3.3 Assumptions and Setting

We simplify the problem by considering only source code generated by csmith [164]. Csmith is a code generator used in many works on compiler correctness [5,61,96,138]. Csmith generates a random C program that is valid and free of undefined behaviors. The absence of undefined behaviors guarantees a well-defined semantic, and thus avoids the generation of meaningless debug traces. We will show that using Csmith generated programs is enough to find novel bugs in the LLVM toolchain.

4.3.4 Solution Overview

All our approaches are based on training unsupervised models on a set of debug traces obtained by programs generated by Csmith. We use two models, one that learns the relationships between the source lines executed in a trace, and the other that learns the mapping between assembly instructions and C code.

Consistency on Source Lines A trace T is associated with a sequence of executed source lines $L(T)$. On this sequence we train a variation of a masked language model using a transformer architecture, we call such network *Source Lines Network* (SLNet). Our hypothesis is that a model trained on this task can identify odd stepping behaviors in debug traces (as in Snippet 4.1). As a matter of fact, [45] has shown that this is a frequent category of bugs, and it is particularly nefarious since a developer debugging a software could see an execution trace that is not consistent with the real execution flow, making it hard to understand what is really happening.

[3] A program is valid if can be compiled by our toolchain and is free of undefined behaviors.

Consistency of the Mapping Assembly-lines A trace T can be seen as a sequence of pairs, each of them mapping a sequence of assembly instructions $A(l)$ to a source line $l \in L(T)$. On these pairs, we train a transformer architecture on the task of identifying the correct mapping between assembly instructions and a source line. We use the term *Mapping Network* (MapNet) to indicate this architecture. An example of a bug found by MapNet is in Snippet 4.2. When stepping on line 8, lldb shows a shift assembly instruction associated with this line (`shll cl, edx`), this shift instruction should be associated with line 6 instead. We remark that no existent approach is able to find this kind of bugs.

```
1 int a, b;
2 int *g_3377 = &b;
3 int main() {
4   short c;
5   char l_3718[7][4];
6   c = 3 > 7 >> a ? 0 : 3 << a;
7   l_3718[6][3] = c;
8   (*g_3377) = l_3718[6][3];
9 }
```

Snippet 4.2. Clang bug 51507, wrong assembly mapped to line 8.

4.4 Architectures Details and Unsupervised Training Tasks

We solve the CDD problem with two architectures, the Longformer [22] used for SLNet and BERT [44] used for MapNet. Both architectures are encoder-only transformers composed of a stack of N identical layers, where each layer is composed of a multi-head self-attention mechanism and a fully connected feed-forward network.

4.4.1 Source Lines Network: SLNet

The SLNet is based on the Longformer architecture which has modified attention that scales linearly with sequence length, thus making it more suitable for processing long sequences of source code lines. In particular, this modified attention consists of a windowed self-attention that reduces time and space complexity by focusing only on local context.

Training SLNet is trained on the Masked Source Language Modeling Task (MSLM) which is a variation of the traditional Masked Language Modeling (MLM). The objective of MLM is to hide a certain percentage of tokens in a sentence and then teach the network to reconstruct the original tokens based on the surrounding context. We adapt this traditional NLP task to debug traces. We do so by hiding a certain percentage m_l of tokens inside one single random line l_j and teaching the network to guess them. Training is done by minimizing the reconstruction error (i.e. cross-entropy loss) of masked tokens. The training process is shown in Figure 4.1.

We use this variation of the original task since we are interested in detecting wrong source line stepping; this behavior manifests in real bugs as a source line that is out of context. By training the network to reconstruct a single source line in a correct trace, we expect that the reconstruction loss of a misplaced source line will

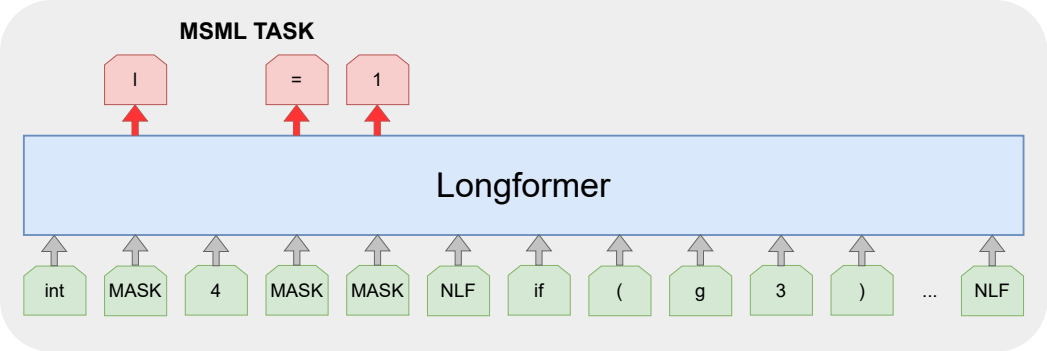


Figure 4.1. SLNet Training on the Masked Source Language Modeling (MSLM) Task.

be high; the network should not correctly predict a line using a wrong context. For this reason, we use higher values for m_l (≥ 0.6) than the ones used in the MLM task. Notice that even with values of masking near or equal to 1.0 is still possible to do meaningful predictions. Consider the sequence `int a=0; int b=0; int c=a+b;` and m_l is 1.0, the masked result is `int a=0; MASK MASK MASK MASK, int c=a+b;`. The model can infer that the line is initializing b, even if it is entirely masked. Note that during the training we do not use labels that identify a trace as bugged or not. Our approach is therefore unsupervised as the analogous task in Bert [44].

Inference We solve CDD using SLNet to compute a score for each function. For each sample, we iteratively mask m_l of the tokens inside each source line and we compute the average reconstruction error. Finally, we take the max of these values to obtain a score for each function. Figure 4.2 shows an example of inference with m_l equal to 0.6 on a bugged function trace composed of three lines. In *Input* 1 three tokens of the first line are masked, while *Input* 2 and *Input* 3 have one and two tokens masked in the second and third line respectively. The network is fed with all of these inputs, one at a time. For each input, we compute the averages of the losses of masked tokens which are 0.0101, 9.4 and 6.6 and we take the maximum as the score. The maximum is 9.4 and represents the score assigned to the function in question. Notice that 9.4 corresponds to the second line, which is obviously misplaced since a return instruction cannot appear in the middle of a function trace.

The computed score can be either used to pick the top k scored function as bug candidates or compared with a certain threshold value.

4.4.2 Mapping Network: MapNet

The MapNet is based on the BERT architecture trained in a cross-lingual fashion [85] by taking as input parallel sentences in two different languages (assembly and C source code).

Training MapNet is simultaneously trained on two tasks: Asm/Source Mapping Prediction (ASMP), and Masked Asm Language Modeling (MALM). Similar to the traditional Next Sentence Prediction Task (NSP), the objective of ASMP training is to have the network predict whether a given sequence of assembly instructions is correctly mapped to a certain source line. We create a dataset for this binary

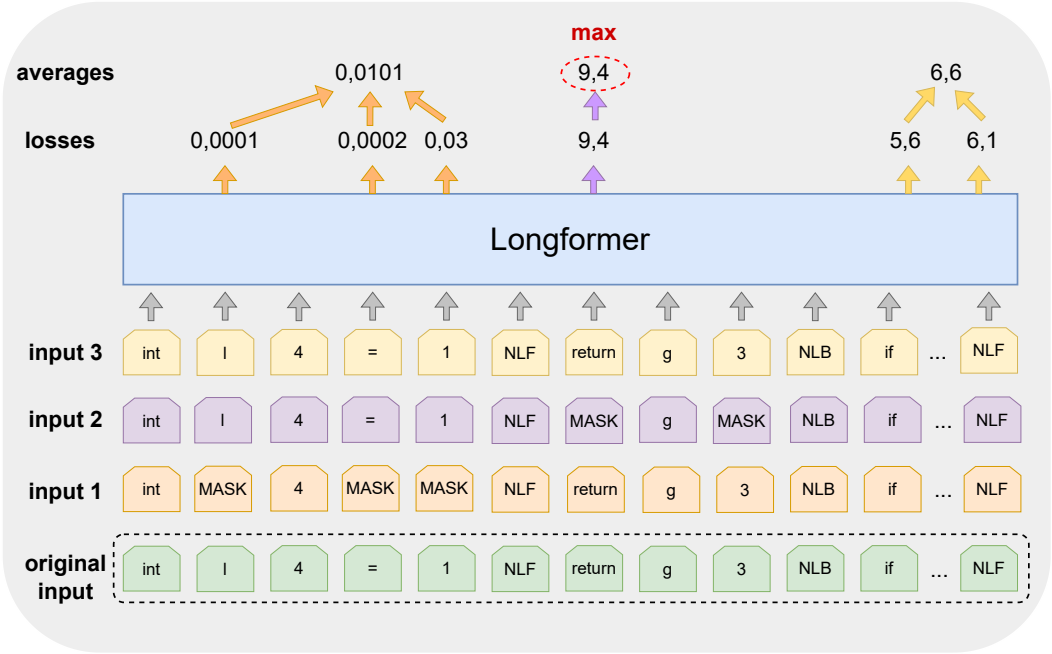


Figure 4.2. SLNet Inference.

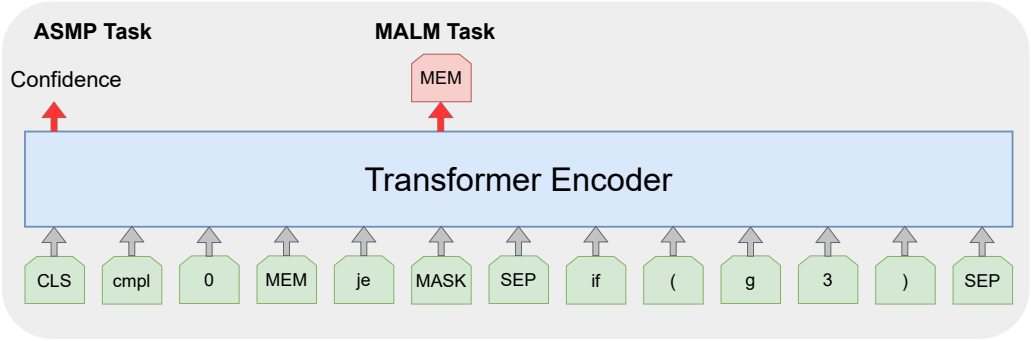


Figure 4.3. MapNet Training on the Asm/Source Mapping Prediction (ASMP), and Masked Asm Language Modeling (MALM) Tasks.

classification task, from the original samples mapping a source line l to assembly instructions $A(l)$. We partition all samples in half; a partition is left untouched while the other partition is used to generate incorrect mapping pairs. The incorrect pairs are created by randomly shuffling the mapped source lines. The MALM task consists in hiding a certain percentage of tokens inside the list of assembly instructions and in teaching the network to reconstruct them. The training loss is the sum of the MALM and ASMP cross-entropy losses. The MapNet training process is shown in Figure 4.3. As for the MSLM task, MALM and ASMP do not use labels that identify a trace as bugged or not. Therefore, we consider our approach unsupervised as the analogous tasks in Bert [44].

Inference We solve the CDD using MapNet to compute a score for each function. For all samples, we use the MapNet to compute the probability that each sample represents a wrong mapping, by taking the confidence value returned by the classifi-

cation layer attached to the hidden state of the CLS token. This is also done in Bert to compute the confidence of having a pair of consecutive sentences. Then, we group all samples belonging to a function aggregating all ASMP confidence values by using the max function. This aggregated value is the score of the function. As for the SLNet, this score can be used to extract the top k samples for analysis or all the ones that are above a certain threshold.

4.5 Datasets

In this section, we describe the datasets used for the experimental evaluation. We consider three distinct sets of programs: one set is used for training and validation, and the other two are used for testing. From these programs, we generate debug function traces. For MapNet, the traces are used to build datasets of samples, each being a mapping pair. For SLNet each sample is a sequence of source lines in a function trace.

4.5.1 Dataset Preprocessing

In this Section, we explain the preprocessing used in our datasets for SLNet and MapNet.

SLNet Preprocessing Given a function trace T_f, we create a sample for the SLNet by first converting all hexadecimal numbers into base ten integers, and then by substituting all numbers greater than a certain threshold (we used 1,000) with a special token DEC. The lines are then tokenized using codeprep [77] removing underscore tokens. Finally, from a sequence of lines $[l_0, l_1, \ldots, l_m]$ we obtain a single string $l_0\, X_0\, l_1 \ldots X_{m-1}\, l_m$ by concatenating the lines. The token used for concatenation X_j, is either: NLF, indicating that l_{j+1} is at a higher line number than line l_j in the original program; or, NLB, indicating that l_{j+1} is at lower line number than line l_j. Practically speaking, we use NLB when the trace jumps backward (e.g., a for loop). An example of a preprocessed SLNet sample is shown in Figure 4.1. We removed all duplicates and all sequences of a single source line.

MapNet Preprocessing For the MapNet samples, we preprocess the assembly by executing a symbolication step (we substituted addresses with variable names, strings, and function names). We then substitute all the memory accesses with a special token MEM, we convert immediate operands into base ten substituting the ones above 1,000 with token DEC. We obtain a single string by concatenating all assembly instructions using whitespaces. We use the same strategy to preprocess numbers in the source line. We then preprocess source line strings as shown in the previous paragraph. Finally, the two strings are concatenated using a special token SEP. An example of a preprocessed MapNet sample is shown in Figure 4.3. We removed all duplicate samples. Moreover, the csmith generated programs contain many assignments of the *csmith_sink* variable and calls to the *transparent_crc* function: we performed a downsampling of samples containing these patterns in the MapNet dataset by taking 2% of them.

4.5.2 Training and Validation Datasets

We used Csmith to generate 43,921 programs and lldb to obtain 200,443 debug function traces. We split the dataset into training and validation (90%-10% split). After preprocessing, the SLNet datasets result in 81,337 traces for training and 7,195 for validation, while the MapNet datasets end with 349,153 and 25,286 mapping pairs. We analyzed the length of the samples in our datasets. For the SLNet datasets, we have that 70% of the samples contain less than 1,024 tokens, while for the MapNet we have that 99.9% of all samples contain less than 512 tokens.

4.5.3 Synthetic Datasets

These datasets were used to evaluate the performance of the model in identifying bugs. Programs inside the synthetic datasets were created as the training ones. Additionally, starting from these programs, we created both synthetic bugged traces, in which we add different types of errors, and bug-free traces, in which traces remain untouched. By observing real cases, we identified three main categories of bugs that were inserted inside the original *function traces*. Given a function trace $T_f : [s_0, s_1, \ldots, s_n]$, synthetic bugs are defined as follows:

- **swap source**: we randomly select two steps $\{s_k, s_l \mid line(s_k) \neq line(s_l))\}$ and we assign $line(s_k)$ to step s_l and $line(s_l)$ to step s_k.
- **swap assembly**: we randomly select two steps $\{s_k, s_l \mid asm(s_k) \neq asm(s_l))\}$ and we assign $asm(s_k)$ to step s_l and $asm(s_l)$ to step s_k.
- **remove step**: we randomly select a step s_j and we remove it from T_f.

We generated 3,983 programs and then we uniformly inserted one of these bugs inside one function trace for 27% of the programs. The insertion of a bug in a trace creates a single bugged sample for the SLNet dataset, while it impacts one or more samples in the MapNet dataset. Therefore, in MapNet we mark as bugged all samples deriving from a bugged trace. After duplicate removal, for the SLNet dataset, we have 6,009 non-bugged samples and 630 bugged samples (Swap Source: 338; Swap Assembly: 149; Remove Step: 143). For the MapNet dataset, we have 90,739 non-bugged samples and 10,653 bugged samples (Swap Source: 5,352; Swap Assembly: 3,093; Remove Step: 2,208).

4.5.4 Real Bugs Datasets

These datasets were created by using real bugs from the LLVM repository. We analyzed 42 bug reports and identified bugs that could exhibit a wrong step behavior or a wrong mapping assembly/source. For each bug, we created a program that generates the reported bugged behavior and obtained a trace using the LLVM toolchain version containing that bug. In the program, we normalized variables and function names using the same naming convention used by csmith. At the end of this process, we obtained 16 different programs. When possible, for each program we created a trace that does not contain the bug by using a patched toolchain. Note that this generates a challenging dataset since the only difference between

bugged/non-bugged traces is the presence of the bug itself; both traces are derived from the same program.

We obtained a dataset for SLNet with 29 traces, 18 bugged and 11 bug-free. For MapNet we obtained 136 samples, 76 bugged and 60 bug-free.

4.6 Experimental evaluation

This Section reports the results of our experimental evaluation on the synthetic and real datasets.

4.6.1 Training, models parameters, and metrics

Since our amount of training data is limited, we use smaller architectures than the ones usually adopted in NLP. We use [145] as a guide for the selection of parameters for smaller transformer models. We test medium models composed of 8 layers, 8 attention heads, embedding size 512 and intermediate size 2048, and small models composed of 4 layers, 8 attention heads, the same embedding and intermediate size of the medium ones. For SLNet we use a sequence length of 1024 tokens, thus truncating 30% of the sequences. We choose this value since it implements a reasonable trade-off between the number of truncated sequences and the computational capabilities of our hardware. The masking rate for the SLNet m_l is a value in $\{0.6, 0.8, 1.0\}$. For MapNet the sequence length is 512 tokens (99.9% of the mapping sequences are above that treshold) and the masking $m_a \in \{0.0, 0.2, 0.4\}$.

We train SLNet for 60 epochs and MapNet for 30 epochs, taking the models with the lowest validation loss. The training uses Adam optimizer with a learning rate of 10^{-5} on 8 A100 GPUs using a batch size of 16 for each device.

Metrics We evaluate the performances of our model in identifying bugs by computing the Area Under the Curve (AUC). This is done by assigning to each function trace one SLNet score and one MapNet score as defined in the inference Sections 4.4.1, 4.4.2. On the synthetic datasets, we compute a single AUC for each class of bug; we do this by selecting a certain category (e.g., Swap Source) and ignoring all the bugged samples belonging to other categories. In this way, we compute the specific performance on a certain category of bug. On the real dataset, we use a single AUC score as bugs do not belong to a specific category.

4.6.2 Results on the Synthetic Datasets

In this Section, we will show the results we obtained with both SLNet and MapNet on the synthetic datasets.

SLNet Figure 4.4 shows the results of SLNet on the Swap Source bug category, which is the only bug category recognized by SLNet. In this case, the SLNet model reaches an AUC of 0.74 when using the medium model with $m_l = 0.6$ during training and $m_l = 0.8$ during inference. This is the model with the best performance; all the small models, as well as the medium models with training $m_l \in \{0.8, 1.0\}$, have worse or comparable performances.

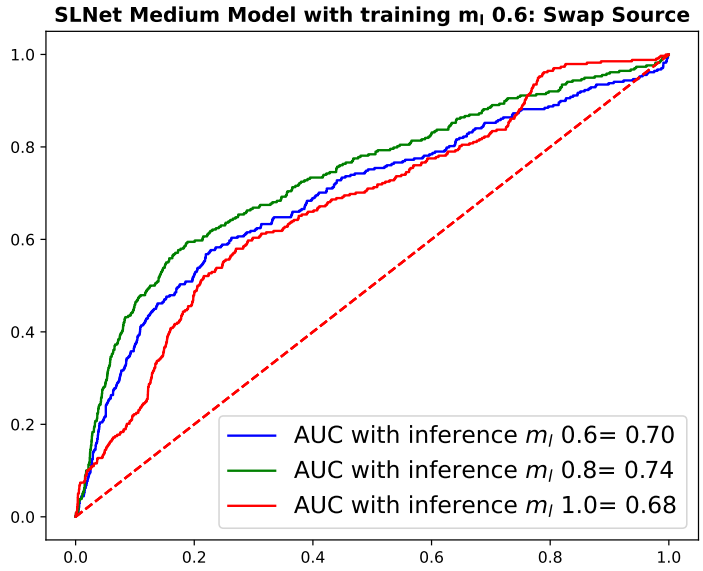


Figure 4.4. Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$.

The ROC curve for the Remove Step bug category of SLNet is in Figure 4.5. The performance of the network is similar to a random classifier. For the Swap Assembly bug category, the ROC is in Figure 4.6; also in this case the network behaves as a random classifier. We believe that SLNet provides random performance on these bugs since the removal of a step rarely impacts the source line trace; the swap of assembly instructions, that leaves intact the source line information, is invisible at the source level.

MapNet The results for MapNet are reported in Figures 4.7, 4.8, and 4.9. MapNet exhibits the best performance when trained with medium model, $m_a = 0.2$, and evaluated with $m_a = 0.0$. The Swap Source bug category (Figure 4.7) is the one with the highest results (AUC 0.89); this is expected as this category of bugs is analogous to the defects inserted in the training task. MapNet is able to discover also the Swap Assembly pattern (Figure 4.9), where a single assembly instruction is misplaced, with acceptable performance (AUC 0.75). Therefore, we expect that it will be able to identify real bugs where the sequence of assembly instructions mapped by the debug info is partially correct (we will show that this is confirmed in the following sections). The worst performance is shown in the Remove Step category (Figure 4.8) with an AUC of 0.62. This is expected as it is far easier to recognize a completely out-of-context assembly instruction, than a missing assembly instruction. MapNet reaches a higher AUC than SLNet on the Swap Source, however, it does so by using a completely different mechanism looking at the mismatch between assembly and source line.

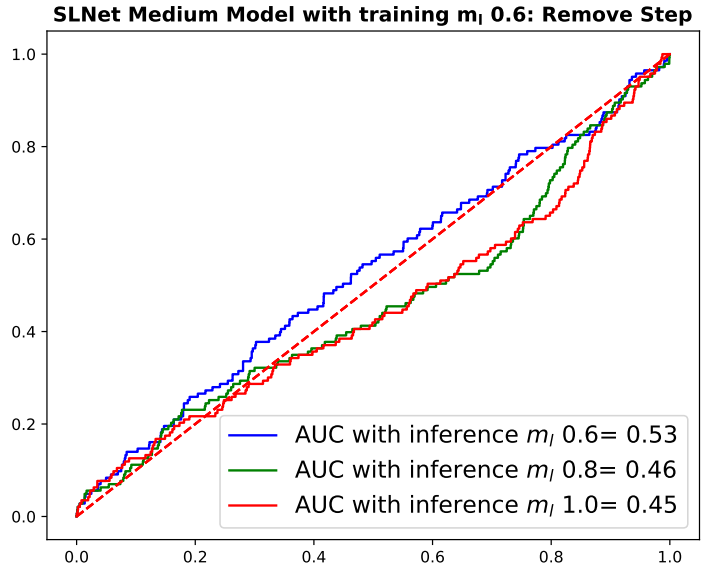


Figure 4.5. Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$: Remove Step bug category.

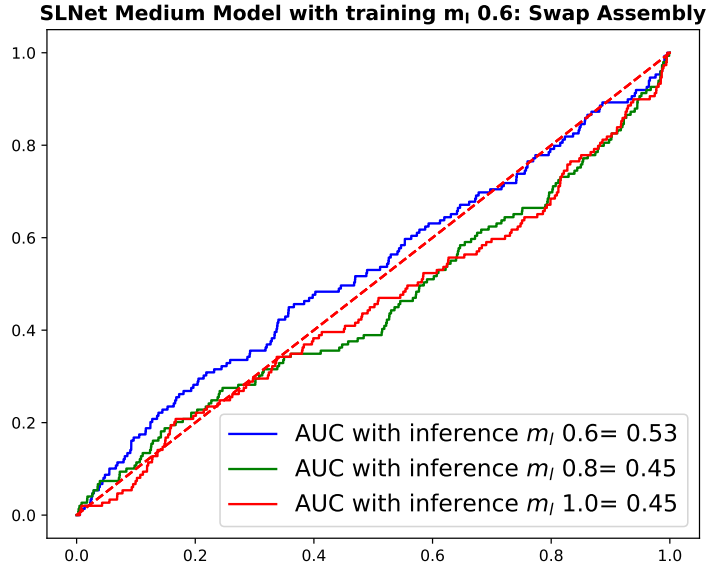


Figure 4.6. Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$: Swap Assembly bug category.

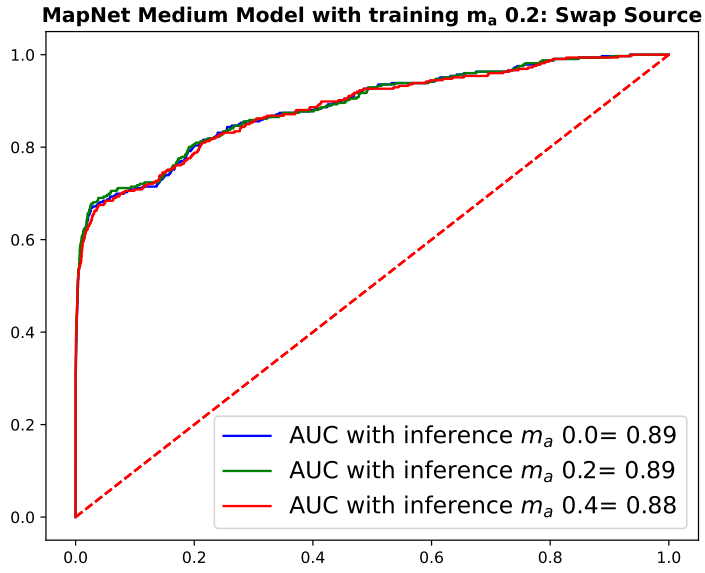


Figure 4.7. Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Swap Source bug category.

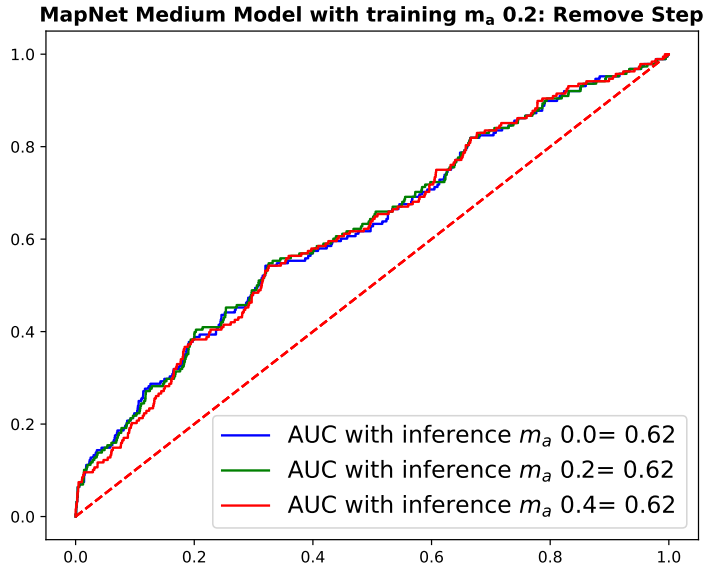


Figure 4.8. Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Remove Step bug category.

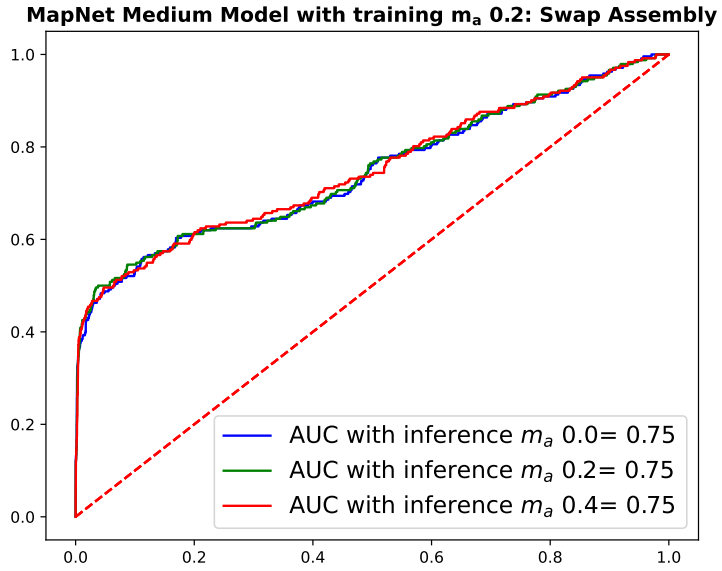


Figure 4.9. Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Swap Assembly bug category.

4.6.3 Results on the Real Bugs Datasets

Given the limited size of the real dataset and the randomness of the masking procedure, we decided to increase the robustness of the results by running the inference procedure multiple times and by reporting the mean AUC value. In order to compute the results on the real dataset, we used the best models according to the synthetic dataset, which are the medium models with training $m_l = 0.6$ and $m_a = 0.2$ for the SLNet and MapNet respectively.

SLNet During inference, we used $m_l \in \{0.6, 0.8, 1.0\}$ and, differently from the synthetic case, we obtained the best performances when $m_l = 1.0$, as shown in Figure 4.10. This is probably due to longer lines and higher variability for csmith source code that makes more difficult for the network to predict them without some suggestions (provided by a certain percentage of non-masked tokens). On the contrary, the real dataset contains small test cases with small lines having a lower degree of randomness, thus the network does not need suggestions to predict them. On the real dataset, SLNet reaches an AUC of 0.81.

MapNet We used $m_a \in \{0.0, 0.2, 0.4\}$ and we obtained the best performances when $m_a = 0.0$, as in the synthetic dataset case (Figure 4.11), the AUC reached is 0.8. This confirms that MapNet is able to identify real bugs.

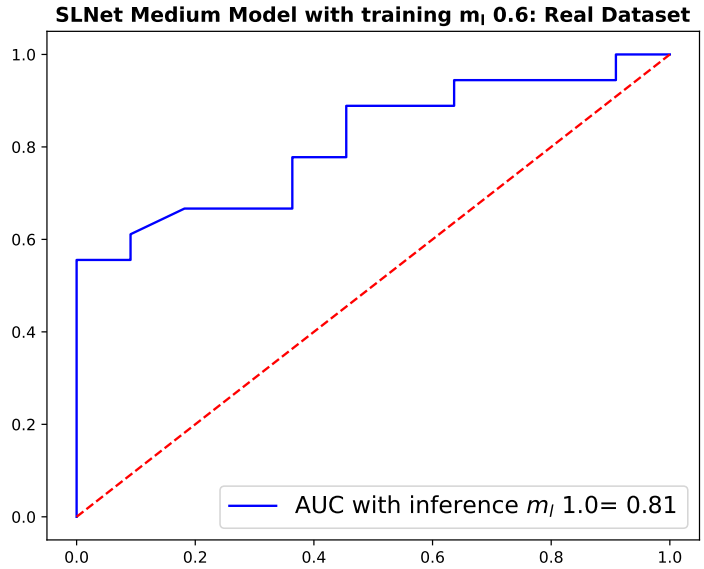


Figure 4.10. Results on real dataset of the SLNet trained with $m_l = 0.6$.

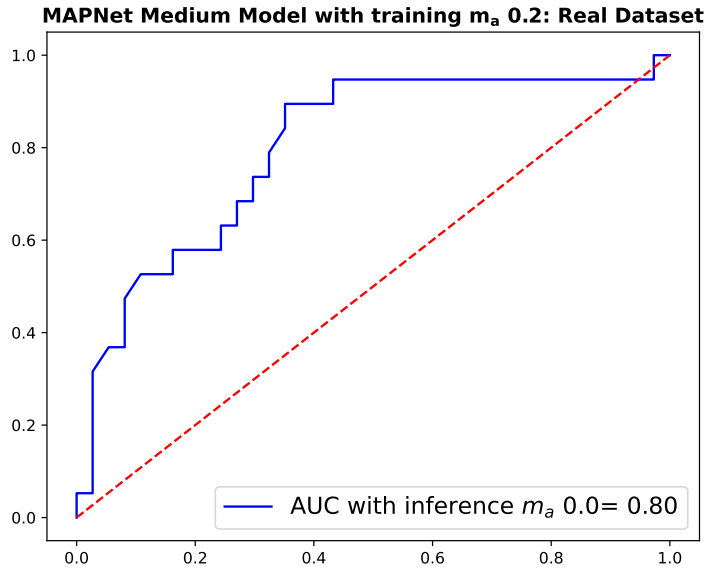


Figure 4.11. Results on real dataset of the MapNet trained with $m_a = 0.2$.

4.6.4 Threshold Analysis

In Section 4.6.2, 4.6.3, we evaluated the performances of our networks by using the AUC metric. The aim of this Paragraph is to show the value of metrics at fixed thresholds. In particular, we compute the precision, recall, and F1 of the most prominent categories of bugs by using our best models, as defined in Section 4.6.3.

	SLNet $\mathbf{m_l = 0.8}$	**MapNet $\mathbf{m_a = 0.0}$**	
Threshold	3.3	0.5	
Bug Category	Swap Source	Swap Source	Swap Assembly
Precision	0.20	0.71	0.55
Recall	0.46	0.57	0.38
F1	0.28	0.64	0.45

Table 4.1. Precision and Recall of SLNet and MapNet with fixed thresholds. The values of m_l and m_a are used during inference.

	Random classifier for source lines	**Random classifier for mapping assembly-line**	
Bug Category	Swap Source	Swap Source	Swap Assembly
Precision	0.053 ± 0.003	0.028 ± 0.002	0.024 ± 0.002
Recall	0.50 ± 0.03	0.50 ± 0.027	0.50 ± 0.031
F1	0.096 ± 0.005	0.062 ± 0.003	0.047 ± 0.003

Table 4.2. Precision and Recall of random SLNet and MapNet with fixed thresholds.

We selected the thresholds to maximize the precision metrics: we want to minimize the number of false-positive bugs triggered and the waste of human time analyzing them. The chosen thresholds are 3.3 and 0.5 for SLNet and MapNet respectively. Results (see Table 4.1) confirm that MapNet can identify bugs with acceptable performances. In fact, MapNet reaches a precision of 0.71 and a recall of 0.57 with the Swap Source bug category, while SLNet reaches a precision of 0.20 and a recall of 0.46 on the same bug category. To have a reference, we also compute these metrics for a random classifier that uniformly flags a trace as bugged or not (see Table 4.2). Since these datasets are highly imbalanced, in some cases the recall is higher than the one obtained by our networks; on the contrary, precisions are very low compared to the ones shown in Table 4.1, thus confirming the effectiveness of our system.

4.6.5 MapNet and SLNet Correlation

We employ the data described in Section 4.7.1, to analyze the possibility of linear correlation between the highest losses obtained by SLNet and the ones of MapNet using Spearman's ρ [170], the results show that there is none to very-weak negative correlation (spearman -0.04 with a p-value < 0.01). This is confirmed by the Kendall Tau coefficient [79] (value of -0.02 with a p-value < 0.01). Our interpretation is that the two networks are able to identify different kinds of bugs, and this is why

we keep them apart in the proposed framework.

4.7 Finding Novel Bugs: Neuro-Debug^2

We integrated our best models, SLNet medium with training $m_l = 0.6$ and MapNet medium with training $m_a = 0.2$, in the Debug^2 framework [45].

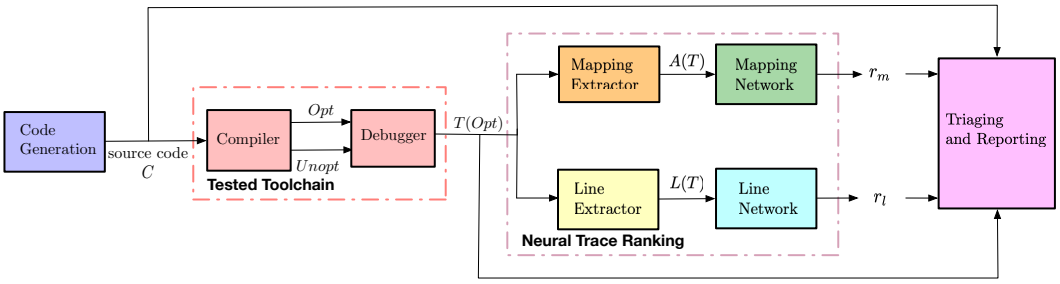


Figure 4.12. Neuro-Debug^2: System Overview

The modified framework is depicted in Figure 4.12 and contains the following modules:

- **Program Generation:** generates the programs that will fuzz the toolchain under test. This module uses csmith as generator. The output is a test program c.
- **Tested Toolchain:** this module contains the toolchain F to be tested. It takes as input a test program c and generates an optimized debug trace using F. Practically speaking, the input program is compiled with optimization level $-Og$ and fed into the debugger to get a set of function debug traces.
- **Neural Trace Scoring:** this module analyses each function trace T_f using the SLNet and MapNet. It outputs a tuple $< T_f, r_l, r_m >$ where r_l and r_m are the scores given by SLNet and MapNet. The Mapping Extractor and Line Extractor transform the execution trace in a sequence of source lines $L(T_f)$ and a set of mapping pairs $A(T_f)$, each object is fed into the respective network.
- **Triaging and Reporting:** this module collects the traces and orders them by their rank. The likely bugs are manually investigated; if confirmed as bugs, they are reduced to smaller examples (this is done automatically using creduce [129]). After the reduction, the confirmed likely bugs are reported to the toolchain developers.

4.7.1 Tests and Novel Bugs

We tested the LLVM version 13.0.0 of 6 April 2021. We used Neuro-Debug^2 to generate 3,983 programs, which, after preprocessing, resulted in 11,431 function traces.

Analysis of the Top Scores We used the data described above to compute a score for each function by using both the SLNet and the MapNet, as described in the corresponding inference Sections 4.4.1, 4.4.2. Our research hypothesis is that a high score for functions correlates with the presence of bugs in debug information. To validate this hypothesis we ordered all the functions by their scores (both for MapNet and SLNet) and we selected the top-scored functions.

Specifically, we analyzed the 40 top-scored functions for SLNet with inference $m_l \in \{0.6, 0.8, 1.0\}$ and MapNet with inference $m_a \in \{0.0, 0.2, 0.4\}$. [4] This leads to a total of 240 traces, 120 for each model, and 40 for each fixed combination of model/parameter. For each batch of 40 traces, we performed a manual analysis to determine how many of these traces are actually bugged or not: if a previously unknown bug was encountered during this analysis we reported it to compiler' developers.

The outcome is reported in Table 4.3. For SLNet, the bug percentage reaches its maximum 50% with $m_l = 1.0$ and it decreases monotonically with the masking. This means that half of the 40 top-scored functions indeed contain a bug. To rule out that this prevalence of bugs was due to chances, we took 40 random functions from our dataset and analyzed it manually noting the number of bugs encountered. From our analysis only 7% of the functions in the randomly sampled set contain a bug. This means that the prevalence of bugs in the 40 top-scored functions for SLNet is 8 times more than random.

When considering MapNet we have that 77% of the traces contain one or more bugs when $m_a = 0.4$; different masking levels give a slight decrease in performance. As for SLNet, to rule out that this effect was due to chance, we took 40 random samples from the MapNet data and analyzed them for the presence of bugs. We found that 27% of the samples contained a bug in the mapping. Note that this number is different from the 7%, reported for the analogous case in SLNet, because in this case we manually verify if each mapping is correct, while in the previous scenario we only verify that the sequence of shown source lines is correct. We highlight that also for MapNet we found almost 3 times more bugs than the ones present in a random sample. This means that highly-scored samples are likely to be bugs, thus human experts could save time in analyzing debug information correctness by using suggestions provided by our system.

Reported Bugs From the bugs found in the previous Section, we sampled a subset that we reported to LLVM developers. Specifically, we took 6 bugs found by SLNet and 6 by MapNet and we verified that they were still present in the last LLVM version 14.0.0 as of August 15th, 2021. Out of the 12 reported bugs, 2 have been confirmed by the developers; the remaining 10 bug reports are pending analysis. In this Section, we discuss one bug found by MapNet and one by SLNet.

A bug found by SLNet is in Snippet 4.3. In this case, lldb shows that the execution steps on line 6 (`int i, j;`); however, this is a variable declaration that should not be shown during the execution. This bug is likely due to an error in the line table (the structure that maps assembly instructions to source lines) produced by clang. We speculate that SLNet flags the stepping on a declaration as unlikely

[4] We discarded the samples that had a length above the thresholds used for truncation.

Network	Bug Percentage
SLNet $m_l = 1.0$	50%
SLNet $m_l = 0.8$	37.5%
SLNet $m_l = 0.6$	27.5%
Random traces from SLNet samples	7%
MapNet $m_a = 0.4$	77%
MapNet $m_a = 0.2$	75%
MapNet $m_a = 0.0$	72.5%
Random traces from MapNet samples	27%

Table 4.3. Analysis of the top 40 scores for SLNet, MapNet and randomly sampled functions. The values of m_l and m_a are used during inference.

```
1 short a;
2 int b;
3 void func_1() { a = 0; }
4 int main() {
5   b = 0;
6   func_1();
7 }
```

Snippet 4.4. Clang bug 51751, wrong assembly mapped to line 6.

since it has not been observed as a normal behavior during training.

```
1  int a, c, e, f;
2  static int *b = &a;
3  short d = 6;
4  void func_15() {
5    for (; c >= 0; c--) {
6      int i, j;
7      *b = f;
8    }
9  }
10 int main() {
11   func_15();
12   for (; d <= 0;) {
13     int g[4];
14     g[e] = &b;
15   }
16 }
```

Snippet 4.3. Clang bug 51512, wrong assembly mapped to line 6.

MapNet discovered the bug in Snippet 4.4; in this case the call of `func_1` at line 6 is wrongly associated to an assembly instruction that set variable a to 0 (`movw $0x0, 0x200b89(rip)`); this assembly instruction should instead be ,mapped to the body of `func_1`. As in the case of Snippet 4.1, this bug is probably the result of the inlining optimization.

Network	Neuro-Debug^2 Bugs	Debug^2 Bugs
SLNet $m_l = 1.0$	20	11
MapNet $m_a = 0.4$	31	2

Table 4.4. Results of the comparison between Neuro-Debug^2 and Debug^2. The values of m_l and m_a are used during inference.

4.8 Comparison with Debug^2 and Limitations

In this Section, we compare our approach with Debug^2, and then we discuss the limitation of our approach.

4.8.1 Comparison with Debug^2

We evaluate the effectiveness of our proposed solution with the respect to the invariants-based approach of Debug^2 [45].

In particular, we take the 31 programs containing bugs detected by the MapNet with training $m_a = 0.2$ and inference $m_a = 0.4$ and the 20 programs containing bugs detected by the SLNet with training $m_l = 0.6$ and inference $m_l = 0.1$ (see Table 4.3). We measure how many times these bugged functions are identified by an invariant violation of Debug^2. We find out that, among the 31 bugs detected by MapNet, only 2 are discovered by Debug^2, while 11 out of 20 bugs discovered by SLNet were identified by Debug^2 as well (see Table 4.4). In our test, Debug^2 is more capable of finding bugs identified by the SLNet rather than MapNet ones. This is expected, SLNet has been designed to detect source lines that are out-of-context, some of these cases are covered by the LineInvariant of Debug^2.

We want to stress that our system is not alternative to an invariant-based approach; a user may analyze bugs found with our solution and identify a general pernicious behavior that could lead to the definition of a new invariant.

4.8.2 Limitations

Manual Analysis Confirming the presence of a bug in an anomalous trace must be done by a human expert. Table 4.3 quantifies how much effective the job of the expert is when working on anomalous traces identified by our system vs. random traces. The manual effort to decide if an anomalous trace is a bug or not requires around 10-20 minutes of time by a human expert. We remark that this manual analysis step is also needed by the other works that find bugs in debug information [45].

Variable Values In addition to the manual analysis requirement, another limitation of our system is represented by the absence of an explicit analysis of variable values. [45] uses two invariants which are based on variables: Scope Invariant and Parameters Invariant. The former checks whether there exists a step where a variable is visible only in the optimized trace, while the latter checks whether the optimized trace contains function parameters values that are not present in the optimized one. Currently, our system is not able to identify mistakes in the values of variables. As future work, we plan to extend our system to directly integrate them.

Correctness of Training Data As in many previous works on neural bug finding [6, 148], we assume that our training data is correct. However, we have no guarantees on such correctness. We argue that this is not a problem. Even if some bugs are so frequent in training that they are not detected as anomalous anymore, this does not jeopardize our approach. As long as there are rare bugs (that could appear in the training data but sparingly) they will be detected as anomalous. The fact that some bugs are rare is reasonable; the contrary would imply that debug information is almost meaningless. Moreover, frequent bugs are likely to be found by humans.

4.9 Conclusion

In this work, we introduced two DNN-based architectures trained for the detection of bugs in debug information attached to optimized binary code. Our results show that the proposed models, namely SLNet and MapNet, are capable of discovering bugs both in synthetic and real datasets. As a result of this study, 12 new bugs in the LLVM toolchain were discovered.

Chapter 5

BinBert: Binary Code Understanding with a Fine-tunable and Execution-aware Transformer

5.1 Introduction

A growing body of literature has demonstrated that Deep Neural Networks (DNNs) can effectively address various binary analysis tasks. DNNs today show state of the art performances for binary similarity [107, 168, 175], compiler provenance [36, 106, 126], function boundaries detection [135], decompiling [55], automatic function naming [41, 53] and others.

DNN designers must decide how to feed binary code to their models. One possibility is to use manually-identified features. This approach requires a domain expert which identifies features of interest forecasting their helpfulness in solving the task at hand. This approach is known to produce problem-specific features and injects a human bias inside the system. Recent solutions automatically transform binary code into a representation usable by the neural network layers.

A common technique is to transform assembly instructions into representational *embeddings vectors*, similarly to what has been done in the Natural Language Processing (NLP) field with the word embedding revolution [144]. Several works [38, 91, 107, 175] proposed refined techniques to transform a single instruction into a vector of real numbers while capturing its semantic (e.g. all vectors of arithmetic instructions are clustered in the vector space). By using this approach, sequences of instructions are transformed into sequences of fixed-size vectors that can be fed into standard DNNs.

A common weakness of all these approaches is *the lack of context*: an instruction is always represented by the same vector, irrespectively of where it appears. However, the semantics of a single assembly instruction is strongly limited (more than a word in natural language), and non-trivial concepts in assembly code are almost always encoded by a sequence of instructions (e.g., loops, swap of variables in memory, calling conventions, etc). Complex semantics, that span sequences of several assembly

instructions, are hardly representable if embeddings of instructions are created in *isolation*; they have to be learned by the neural architecture using the embeddings.

Recent works [121, 122] overcome this limitation by using a transformer [149] based architecture that operates on sequences of assembly instructions. This enables them to embed entire assembly functions taking into consideration the instructions' context. These systems have not been proposed and tested as instruction embedding techniques but as solutions to specific problems (Trex [121] for binary similarity and the Stateformer [122] for type inference).

5.1.1 Execution-aware Binary Code Interpretation

Code serves as a form of communication between humans and machines, possessing a dual nature. One aspect represents the syntactic and semantic meaning that can be inferred from its *static form*, while the other aspect lies in its ability to be *executed*. The full understanding and appreciation of an Instruction Set Architecture (ISA) can only be achieved through the execution of code (e.g., the dependencies introduced by `RFLAGS` in X64). Additionally, sequences of instructions that have the same meaning but a different syntax can be easily identified when they are executed.

Surprisingly, almost all embedding techniques we are aware of, only consider the static aspect of binary code. Notable exceptions are the aforementioned [121, 122] in which the execution is embedded by training the models on the assembly instructions of a function and on the values of CPU registers obtained by a concrete execution with random inputs. One could argue that such an execution approach is prone to the noise and the limited significance of a random execution.

5.1.2 Expressive Power and Fine-tunable Models

Oddly, the existent instructions and function embedding models are proposed and tested on a single problem ([121], [122], [107], [38]). This is limiting as the expressive power of an embedding model can only be assessed when the model is tested on different tasks. With the current body of knowledge, there is uncertainty on whether and how the proposed embedding models generalize to different tasks or not.

The only exception is Palmtree [91], which proposes an assembly instruction model tested on a few tasks. However, a glaring limitation of [91] is that their embedding model is frozen and not *fine-tuned*.

We argue that an assembly model has to be tested using the fine-tuning paradigm. That is, the model is first pre-trained on a large corpus of assembly code using several tasks. During pre-training, the model learns a general semantics of assembly sequences that is context and execution aware. Then, the pre-trained model is used as part of a DNN that solves a specific *downstream task* (e.g., compiler provenance, function similarity, and others). The DNN, including the assembly model, is retrained end-to-end on a small amount of problem-specific data during the fine-tuning process. This paradigm is state-of-the-art for NLP and works well also if the fine-tuning dataset is small. This is especially useful for binary analysis tasks where creating a labeled dataset requires expensive manual effort.

5.1.3 Our proposal: BinBert

In this work we introduce *BinBert*, a fine-tunable assembly code model based on a transformer encoder that is execution-aware. To inject execution awareness into our model, our idea is to symbolically execute snippets of assembly code. Specifically, we use a symbolic execution engine that transforms sequences of assembly instructions connected by a data-dependency relationship (the strands introduced in [42]) into sets of semantically equivalent symbolic expressions. These expressions are a functional representation of the input-output relationship of the strand. We designed a novel pre-training process that forces BinBert to learn the correct matching between an assembly sequence and an equivalent symbolic expression and to translate assembly code into symbolic expressions and vice-versa. Our intuition is that symbolic expressions are more useful than using randomised concrete executions as they do not suffer from the same level of noise.

We train BinBert on a new large dataset[1] of assembly sequences and symbolic expressions derived from symbolic execution, obtaining a general-purpose assembly code model. Our model is able to create representative embedding of single instructions, as well as to generate representative embeddings of sequences of instructions that could be either snippet of assembly code or entire functions. We remark that symbolic execution is needed only in the pre-training phase; no code execution is required while using the model for inference tasks.

We tested BinBert on a multi-task benchmark for binary code understanding that we built. Tasks in the benchmark range from intrinsic ones, aimed at evaluating how the pre-trained BinBert captures the semantic of instructions and sequences, to extrinsic downstream tasks, in which we fine-tune BinBert for problems on assembly sequences and binary functions. In all our experiments BinBert raises the performance bar outperforming the current state-of-the-art (including PalmTree [91]) and specific solutions created for the binary similarity problem.

In summary, this work provides the following contributions:

- a novel training task that makes the training of an assembly code model execution-aware by using symbolic expressions derived from the symbolic executions of assembly snippets;
- BinBert, a pre-trained execution-aware transformer model for X64, that can be plugged into DNNs for binary analysis. The model has been pre-trained on a 26 GByte dataset. We release the model, the code used to train it as well as the dataset;
- the first multi-task benchmark designed to test the binary code understanding of assembly models. The benchmark is composed of well-known tasks selected from the literature for their relevance, and two novel tasks (strand recovery and execution) for the semantic understanding of assembly sequences;
- an in-depth performance evaluation of BinBert based on our benchmark that shows how execution awareness improves the performance of an assembly model. To the best of our knowledge, we are the first to thoroughly test the impact

[1] Size-wise our dataset is larger than the original dataset used to train Bert [44].

of the fine-tuning paradigm on assembly representation learning. We show that, as already shown in the NLP field, the pre-training/fine-tuning approach has a positive impact on all downstream tasks. As a consequence, BinBert outperforms the current state-of-the-art instruction embedding techniques.

BinBert has been developed by addressing some of the research gaps identified in our systematization effort, as presented in Chapter 3. Specifically, we considered the following issues:

- The need to explore the full potential of automatically learned features: To address this, we built BinBert as an unsupervised binary code embedding model based on a transformer encoder.
- The improper use of the fine-tuning approach: Most existing works evaluate their models on a single downstream task. In contrast, our BinBert model is a general-purpose model that can be fine-tuned for multiple downstream tasks (see 5.6).
- The scarcity of works based on execution information: We pre-trained our model using symbolic execution and evaluated the contribution of such information on downstream tasks in our experimental study (see Sections 5.3.3 and 5.6).
- The lack of comparisons with standard architectures and pre-training tasks: We compared our model with solutions based on standard networks (e.g., SAFE [107]) and traditional NLP pre-training tasks, such as the MLM task (see 5.6).
- The use of tokenization strategies without comparison or rationale: We evaluated different tokenization strategies in Section 5.6.4.

5.2 Background

In this section, we introduce the general theoretical concepts behind the instruction embedding techniques and focus on the current state of the art. Afterwards, we detail weak points and gaps in current solutions, discussing how these influenced our proposal.

5.2.1 Instruction Embedding Models

An instruction embedding model takes as input an assembly instruction i from a vocabulary V of size d and it returns a vector of real numbers $e(i) = \vec{i} \in \mathbb{R}^n$, n is the embedding size (typically $n \in [128, 1024]$). The vector $\vec{i}$ is a *dense representation* of the instruction i.

In the simplest embedding scheme a random matrix M of size $\mathbb{R}^{d \times n}$ is created, each instruction is mapped univocally to a row of M. A sequence of instructions $I = [i_0, i_1, \ldots, i_m]$ is converted into a sequence of vectors $e(I) = [\vec{i_0}, \vec{i_1}, \ldots, \vec{i_m}]$ using a lookup mechanism. This sequence is fed into the task-specific DNN A. The matrix M is usually *trainable*: its elements are trainable weights and are modified during the training of A.

The groundbreaking idea of the embedding models is to generate the embedding matrix M with a neural network Emb, formally speaking $M = Emb(A)$. The network Emb is trained in an unsupervised way on a corpus C of data. This corpora

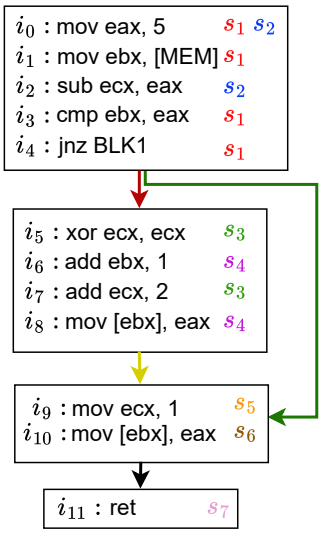


Figure 5.1. CFG of an imaginary function in x64 assembly. For each instruction we indicate with sx the block level strand to which it belongs.

C is composed of sequences of assembly instructions extracted from selected binaries. Usually, the *distributed representation learning tasks* used by instruction embedding models are, apart from minimal modifications, the ones used in NLP by solutions such as word2vec [144], GloVe [123], fastText [26], pv-dm [86]. The common goal is to train Emb to produce an embedding vector that contains enough information to predict a masked instruction from its context in C. Most of the novelty of the instruction embedding system is in the preprocessing of instructions and in the definition of the assembly sequences composing C.

Preprocessing of Assembly Instructions

Assembly language and natural language are distinguished by the wide difference between the vocabulary size d. A natural language is usually composed of hundred thousands different words, while the number of possible distinct assembly instructions is much more. Consider the X64 ISA, a mov instruction can use 64 bits to express immediates, offsets, and memory addresses, thus there can be 2^{64} different instructions that just move a value in a certain register. This makes raw assembly instructions impractical: a large vocabulary is discouraged [34] as it worsens the problem of out-of-vocabulary word (OOW) [70].

Moreover, the exact value of an immediate is largely useless in a static analysis setting (e.g. a memory address of an unknown memory layout) [107]. To ameliorate this problem, a lot of effort has been devoted to instructions preprocessing [46, 49, 91, 107, 121, 175]. The standard of the field is to substitute all memory addresses and immediates above a certain threshold value with special symbols (e.g. `IMM`). Another design choice is whether to consider the entire assembly instructions as a token (used in [107]), to split the assembly instructions into several tokens by separating opcodes and operands (used in [46]) or to use a more fine-grained split strategy [91]. Interestingly, no one used automatic tokenization such as WordPiece [158] that are standard in NLP.

Extraction of Assembly Sequences

A key point is how to extract the sequences from the binary, as this defines the context in which an instruction appears. The context for instruction i_x is composed by k instructions appearing before/after i_x in C. Used extraction strategies are:

- Linearized Control Flow Graph (CFG) [2]: In this case each sequence is a linearization of a CFG (commonly the one provided by a disassembler) [107]. Blocks of the CFG that are not logically related could be sequentially placed in the linerization, and thus an instruction will see a noisy context. An example is the linearization of the CFG in Figure 5.1 induced by instructions numbers: the context of instruction i_8 contains instructions i_9, i_{10} that are not causally related. This injects noise into the learning process.
- Control Flow Graph (CFG) /Interprocedural Control Flow Graph (ICFG) [3]: the sequences are extracted from the recovered CFG/ICFG. This is done either by using a random walk strategy ([46]), or by taking as a sequence a single block [175]. The idea is to have a sequence that respects the logical control flow of the examined program, removing the source of noise highlighted in the previous strategy. We argue that this technique does not completely remove the presence of extraneous instructions in the context. Take the sequence of instructions i_5, i_6, i_7, i_8 in Figure 5.1, the context of instruction i_7 contains i_6 and i_8 that are not causally related.

Transformer Based Solutions

Recent instructions embedding models are based on transformers. In this case the embedding network is not a single matrix, but a complex DNN that is able to transform a sequence of instructions into vectors. The considerations above also applies to these solutions, they have to decide how to extract assembly sequences and how to preprocess such sequences.

PalmTree PalmTree [91] is a transformer-based instruction embedding model that has shown state-of-the-art performances beating all the other embedding models on several tasks. Instructions are divided into tokens using a fine-grained strategy with manually made regexs. The model is trained on pairs of instructions taken from the corpora C. PalmTree uses the standard Masked Language Modeling (MLM) of Bert and two novel tasks: the Context Window Prediction task (CWP) in which the network has to recognise if a pair of instructions is taken from the same context or not, and the Def-Use Prediction task (DUP) in which the network has to recognize if there is a data dependency between instructions. Once trained, the model is used as an instruction embedding model: a sequence of instructions is embedded by applying separately the PalmTree model to each instruction.

[2] A Control Flow Graph (CFG) represents the flow of control in a function and shows the order in which instructions are executed. The CFG is composed of a set of basic blocks (where each block represents a sequence of instructions) connected by edges (where each edge represents a conditional or an unconditional branch.)

[3] An Interprocedural Control Flow Graph(ICFG) is an extension of the traditional CFG since it covers multiple functions within a program.

Trex and Stateformer Trex and Stateformer are two solutions that use micro-execution traces and transformer architecture. Both utilize similar pre-training strategies and have similar architectures but they are used to solve different tasks: Trex is used for function similarity, while Stateformer is used for variables type recovery. The transformer in these solutions is fed with five sequences: the assembly code sequence, the micro-trace value sequence and additional information sequences that specify the order of instructions, architecture, and the position of opcodes. Regarding assembly sequence, the authors treat all symbols as tokens, including punctuation. To generate embeddings for the micro-trace values, Trex uses a Bi-LSTM, while Stateformer uses a Neural Arithmetic Unit. In terms of pre-training, Trex focuses on predicting masked codes and values in micro-traces, while Stateformer is trained on predicting micro-trace values and whether a particular instruction is executed in a trace. Differently from PalmTree, instruction embeddings are not produced in isolation.

5.2.2 Weak Points and Gap Analysis

We can now identify weak points of previous solutions and gaps in the current body of knowledge. From such analysis, we derive the research directions of our solution.

Noisy or Absent Execution Information

The execution of assembly makes clear concepts that would be covert (or unavailable) from its static representation. Apart from the `RFLAGS` example mentioned in the introduction, consider again the sequence I_0 : `lea eax, [ebx * ecx + edx]; mov edi, eax` such sequence is semantically equivalent to I'_0 : `imul ebx, ecx; add ebx, edx; mov edi,eax`. The semantic equivalence could be easily discovered if information taken from the execution is inserted in the pre-training tasks. Unfortunately, current solutions either neglect this aspect ([46, 46, 91, 175]) or use random executions as proxy for the semantic of the function [121, 122]. The ones that use a random execution strategy do not quantify the impact of this information with an ablation study. Additionally, in some cases, random execution is not sufficient to understand the semantics of a binary code snippet. For example, consider the sequence `mov ecx, 5; mov ebx, 4; cmp eax, ebx; cmove ecx, ebx`: if random execution does not include a value of 4 for `eax`, the result for `ecx` will always be 5, leading the network to incorrectly assume that its value is a constant. Instead, a symbolic expression of this sequence would be `If eax eq 4 then 4 else 5`, making explicit all cases.

RD 1: Design an embedding model that takes into account the execution of code using symbolic execution. The impact of execution-related information has to be quantified with a specific ablation study.

No End-To-End Retraining (Fine-tuning)

The use case of almost all the known instruction embedding models is to train the network for a single problem, or to test it on multiple problem without retraining

it [4]. We advocate for testing the benefit of fine-tuning. During fine-tuning, the pre-trained encoder *Enc* is trained with the network *A* end-to-end on a problem specific labeled dataset. The back-propagation algorithm optimizes the internal weights of network *A* and *Enc*; this optimization modifies the weights of *Enc* so that the knowledge learned during pre-training is applied to the downstream task. As a matter of fact, when *Enc* is a transformer, *A* is usually a linear classifier or another simple neural network, since most of the work is performed by *Enc*.

No one has extensively studied how this paradigm copes with solving different goals on assembly language, goals selected to test the semantic and syntactic comprehension of assembly code. This interesting gap of the current body of knowledge gives us a new research direction:

RD 2: Design an embedding model that can be trained end-to-end on a specific task, transferring the general knowledge learned during pre-training. The model has to be evaluated on a multi-task benchmark designed to thoroughly test the syntactic and semantic understanding of the assembly language.

5.3 The BinBert Solution

In this section we describe BinBert. We first give an overview of the system, briefly describing the transformer architecture [149]. We then give the details of the innovative aspects of BinBert.

5.3.1 Overview

The neural architecture of BinBert is the standard transformer encoder [149] used by Bert [44]. A transformer encoder processes sequential data using an attention mechanism, which allows for both the creation of more informative embeddings (by focusing only on relevant parts of the sequence) and good performances (the attention mechanism is implemented using matrix multiplication that is highly parallelizable on GPUs). More specifically, a transformer encoder is composed of N identical layers stacked one on top of the other, where each layer consists of two sublayers: a multi-head self-attention mechanism and a fully connected feed-forward network. Practically speaking, a sequence of n tokens is transformed into a sequence of n latent vectors (one for each token) with a mechanism that we will explain in Section 5.3.4; this sequence is fed into the initial layer of the encoder. Each other layer takes as input the hidden state token vectors returned by the previous layer. The output of the encoder is a sequence of $n + 1$ embedding vectors: one for each input token and a special embedding for the entire sequence (the `[CLS]` vector described in Section 5.3.4).

To avoid the vocabulary inflation generated by the use of raw assembly instructions, BinBert substitutes memory addresses and immediates above a certain threshold with special symbols. Moreover, BinBert splits a single assembly instruction into several tokens using WordPiece [158].

[4] [91] explicitly states that the downstream tasks have been evaluated without fine-tuning since these tasks were implemented in Tensorflow 3 while their model was implemented in pyTorch.

In BinBert we decided to completely remove the noise given by instructions that are contextually related but have no logical relation (see Section 5.2.1) by extracting sequences representing *strands* [42]. Strands are sequences of causally related instructions computing the values of a certain variable. In this way, the context of an instruction never contains extraneous instructions introduced by compiler optimizations. We symbolically execute each strand to extract a set of symbolic expressions; these expressions will be used in our training tasks as a means to inject execution-related information into the pre-training.

During pre-training, BinBert learns the matching between symbolic expressions and strands (this is done using positive/negative pairs); at the same time, samples are partially masked forcing the model to guess the masked tokens by also learning a translation between symbolic expressions and assembly strands.

5.3.2 Instructions Preprocessing and Assembly Sequences Extraction

This Section will describe the instruction preprocessing rules that we have adopted together with the sequence of assembly instructions that we have used for training our model.

Instructions Preprocessing

We preprocess each assembly instruction substituting immediates above a threshold (5000 in our experiments) with the value `IMM` (the same is done for offsets and memory addresses). We use the special symbol `MEM` in case of jumps. We use a threshold-based approach as small immediate values are likely to carry informative content (comparison with small constants in branches and loops, PC/stack relative displacements that identify variables in memory). All immediates/offsets are converted to decimal format. For call instructions, we distinguish if the called function is user-defined or belongs to libc. For user-defined functions, we substitute the called address with `func` (our system is usable on stripped binaries). If it is a call to libc, we substitute the address with the function name (e.g., `call printf`), since external symbols cannot be stripped. Indirect calls are left untouched.

After preprocessing, each instruction is tokenized using WordPiece [158]. The latter uses a probabilistic approach to learn how to tokenize instructions in a way that minimizes the vocabulary size and the OOW problem. Contrarily to manually made regexes, WordPiece automatically learns how to split complex opcodes (as an example `cmovz` will be split in `cmov` and `z` helping the model in understanding the relationship between the cmovX family of X64). We use WordPiece also on symbolic expressions, This provides a uniform tokenization mechanism and vocabulary for the two distinct languages (asm/sym. expr.) used for BinBert.

Assembly Sequences

In BinBert we use the concept of strands to extract the sequences of assembly instructions on which our model is trained. This does not mean that BinBert cannot be fine-tuned and used on CFG blocks or entire functions as our experiments will show.

A strand, originally defined in [42], is a slice of a CFG block constituted by all the instructions that are connected by def-use dependences. More specifically, we consider as an output variable of a block a memory location or a register on which the last operation is a write or the check of a jump. Starting from this variable we construct a *backward-slice* of the block including all the instructions from which the value of such variables depends. To make the concept clear, consider the example in Figure 5.1; the first block contains two strands S1 and S2: S1 is composed by instructions i_0, i_1, i_3, i_4 that influence the `RFLAGS` register later checked by i_4; strand S2 is composed by i_0, i_2 which define the value of variable `ecx`. Other examples of strands are in the figure. We enrich the original definition of strand, by considering as a single strand all the instructions that prepare the input values for a call. In this case, the strand will be constituted by all the aforementioned instructions and the call instruction itself. Therefore, we build our training corpora C by extracting the CFGs in binary, and then decomposing all the blocks in strands. The strands will be the basic sequences in C.

This decomposition has several advantages: it completely removes the noise introduced by instructions that co-occur in the same context only for compiler optimization reasons; the model learns the entire *"causal context"* of an instruction so it is able to see long dependencies among instructions.

5.3.3 Symbolic Execution

In BinBert we employ symbolic execution to convert each strand into a representative symbolic expression. The symbolic execution engine operates on strands of assembly instructions before preprocessing (as it requires the actual values of immediates and memory locations). The engine is built on angr [136]. During the execution, we consider variables on which the strand's first operation is a read as inputs, and variables on which the last operation is a write as outputs.

Each time the strand writes to a variable (either a memory location or a register), we express the written value using a symbolic expression. When a variable is read, it may either be an input (no one has written to it) or contain a symbolic value. If the variable is an input, we set its symbolic value to its address or register name (as example, for `mov eax, [rbp+4]` we have `eax=*(rbp+4)`, for `mov eax, ebx` we have `eax=ebx`).

The symbolic expression obtained with this process may have one of three possible forms:

- If the strand computes the value of a certain variable, the symbolic expression describes the value in the output variable as a function of the strand inputs. For example, in the first row of Table 5.1, we have the symbolic `rcx=-1 add (0 Concat rsi[1:0])`. This expression is for the output variable `ecx`: the `and` operation extracts the two least significant bits from `rsi`, the result is extended to 64 bits, and then decremented by the `rep`. The expression only contains the extended register of X64, which is a design choice that we discuss later in Section 5.3.4.

- If the strand computes the predicate checked by a conditional branch, then our symbolic expression will be the comparison of the jump condition with a

Strands	Symbolic Expressions
`mov ecx, esi`	`rcx = -1 add ( 0 Concat rsi[1:0] )`
`and ecx, 3`	`rdi = 1 add rdi`
`rep stosb byte ptr [rdi], al`	`*(rdi) = al`

`mov rax, qword ptr [rbp - 168]`
`mov eax, dword ptr [rax + 24]`
`test al, 2`
`jne MEM`
	`0 Concat *(*(rbp add -168) add 24)[1:1] ne 0`

`mov esi, IMM`
`mov rdi, qword ptr [rbp]`
`call fprintf`
	`fprintf(*(rbp), IMM)`

Table 5.1. Examples of symbolic expressions obtained by different strands.

symbolic expression of the value used in the predicate. For instance, consider the second row of Table 5.1; in this case, the symbolic expression is compared with 0 using the not equal (ne) predicate.

- Finally, if the strand computes the arguments used by a call instruction, our symbolic expression will be the call to the specific function (including the symbolic name if it is a libc call), where all the arguments are substituted by the symbolic expressions of their values. We extracted function arguments following the X64 calling convention. An example of a call expression is in the third row of Table 5.1.

From a single strand we may obtain multiple symbolic expressions. For example, in the first row of Table 5.1, the `rep stosb` instruction repeatedly places the content of `al` into the memory pointed by `rdi` for `ecx` times, decrementing `ecx` and incrementing `rdi` with each iteration. This means that the strand has three output variables (a memory location and two registers), and each one will have its symbolic expression. We will call this set of symbolic expressions the *representative set* of the strand.

Preprocessing and Tokenization of Symbolic Expressions The symbolic expressions of each strand are preprocessed similarly to assembly. Large numerical constants are substituted with the special symbol `IMM`, for floating point numbers all the digits after two decimals are truncated. The symbolic expressions are tokenized using WordPiece. During the preprocessing phase, we substitute the name of all registers to their extended form (i.e., we use `rax` instead of `eax`). We do so to help the network in understanding the relationships between names used to address different parts of the same logical register; this step is not applied while preprocessing assembly instructions (i.e., we leave `eax` in the strand).

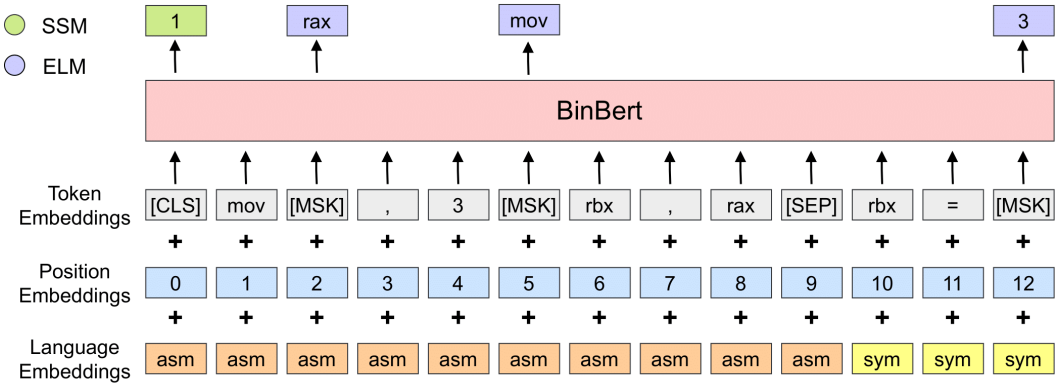


Figure 5.2. Example of BinBert pre-trained on the Execution Language Modeling (ELM) and on the Strand-Symbolic Mapping (SSM) tasks.

5.3.4 BinBert Input Representation and Pre-Training Tasks

BinBert is fed with pairs <strand, symbolic expression> and pre-trained on two tasks that we name Execution Language Modeling (ELM) and Strand-Symbolic Mapping (SSM).

Input Representation

A BinBert input consists of a tokenized strand-symbolic expression pair. Two special tokens are added to each sample: `[SEP]` is used to distinguish between assembly and symbolic expression, and `[CLS]` is prepended to all samples. The hidden state of the `[CLS]` token in the last hidden layer is typically used to obtain a latent vector representation of the entire sequence [44] (see Figure 5.2).

We employ dynamic padding, padding shorter sequences with the special token `[PAD]`, while truncating sequences longer than a threshold (we use 512 in our experiments). Before processing by the transformer architecture, each token is converted into a vector using the lookup mechanism described in Section 5.2.1. Token embeddings are then summed with both position and language embeddings [44, 85]. Position embeddings enable the model to be aware of sequence order, while language embeddings help distinguish between assembly code and symbolic expressions (see Figure 5.2).

Pre-Training Tasks

The first task is **Execution Language Modeling** (ELM). The objective of ELM is to mask a certain percentage *mp* of tokens in the input pairs and have the network predict the original ones. As in Bert, the tokens to be predicted are either substituted with a special token `[MASK]`, replaced with a random one, or left untouched with probabilities of 80-10-10, respectively. Figure 5.2 shows an example in which the tokens `rax`, `mov`, and `3` are masked, and the network attempts to reconstruct them as output. Note that in order to reconstruct a token contained in a strand, the network must pay attention to both the strand and its corresponding symbolic expression (the same applies for a token in the symbolic expression). This means that to predict

the value 3 for register `rbx`, the network is forced to understand the data flow in the strand, thus making the semantic of each strand instruction explicit. As in [44], a linear layer is added on top of the last hidden layer, specifically to the hidden states of the masked tokens; this layer will guess the masked tokens.

Mathematically speaking, we have a dataset $\mathcal{D}$ of <strand, symbolic expression> pairs $I = [t_1, \ldots, t_n] \in \mathcal{D}$, where each token t_i belongs to a vocabulary $V \in \mathbb{R}^d$. We feed each pair to BinBert to obtain context-aware hidden vectors $H = Binbert(I) = [\vec{h_1}, \ldots, \vec{h_n}]$ as output. Now, consider a function $msk(I, m_p)$ that randomly masks m_p percent of tokens in I. The goal of the network is to predict the probability that a token t_i corresponds to the target word $\hat{t}_i$ using the softmax function:

$$p(t_i = \hat{t}_i|I) = \frac{e^{\vec{w}_{\hat{t}_i} \cdot \vec{h_i}}}{\sum_{k=1}^{d} e^{\vec{w}_{\hat{t}_k} \cdot \vec{h_i}}} \tag{5.1}$$

where $\vec{w}_{\hat{t}_i}$ is the weight vector of the linear layer for word $\hat{t}_i$. The loss of the ELM task is the cross entropy loss:

$$\mathcal{L}_{ELM} = -\sum_{I \in \mathcal{D}} \sum_{t_i \in msk(I, m_p)} \log p(t_i = \hat{t}_i|I) \tag{5.2}$$

The second task is the **Strand-Symbolic Mapping** (SSM), in which the goal is to predict whether the symbolic expression is from the set of expressions representative of the strand (see Section 5.3.3). To solve this task, we create both negative pairs by associating a strand with a random symbolic expression and positive pairs in which the symbolic expression is taken from its representative set. The ratio between positive and negative pairs is 50:50. An example of a positive pair can be seen in Figure 5.2: the input strand computes the value 3 for register rbx, as stated by the corresponding symbolic expression. We believe that, with this task, the network is forced to learn the matching assembly snippets and symbolic expressions. This task is built by using a linear layer on top of the hidden state of the `[CLS]` token in the last layer that will be used to classify a pair as negative or positive. In mathematical terms, the goal of this task is to evaluate the probability that the output label is one, i.e., the symbolic expression correctly computes a value in the strand:

$$p(y = 1|I) = \frac{e^{\vec{w_1} \cdot \vec{h}_{[CLS]}}}{e^{\vec{w_0} \cdot \vec{h}_{[CLS]}} + e^{\vec{w_1} \cdot \vec{h}_{[CLS]}}} \tag{5.3}$$

where $\vec{w_0}$ and $\vec{w_1}$ are the weight vector of the linear layer for label 0 (negative pair) and 1 (positive pair) respectively. The loss $\mathcal{L}_{SSM}$ of the SSM task is the standard cross entropy loss.

The final loss on which BinBert is trained is the sum of the losses of the two tasks described above:

$$\mathcal{L} = \mathcal{L}_{ELM} + \mathcal{L}_{SSM} \tag{5.4}$$

5.4 Evaluation Tasks

The proposal of a new assembly model necessitates extensive experimental evaluation. In the field of NLP, standard multi-task benchmarks [153] are used to evaluate language models. However, equivalent benchmarks for binary code do not yet exist. Therefore, we designed our own benchmark by selecting several tasks at the levels of strands, CFG blocks, and functions. This approach tests BinBert on sequences beyond mere strands.

5.4.1 Intrinsic Tasks

The intrinsic tasks [28] directly use the embeddings produced by BinBert; there is no fine-tuning, and the embeddings are not used as input for other models.

Opcode Outliers

In the **opcode outlier** task we are given a set of five instructions. Four of these instructions belong to the same semantic class, and one is an outlier. For example, if the set is `add eax, ebx; sub ebx, ecx; imul ecx, edx; add eax,5; call printf;`, the last one is an outlier. The network has to predict which is the outlier among the five instructions.

Strand Similarity

In the **strand similarity** task we compute the embeddings of given strands with BinBert and use them to discover semantically similar strands. Two strands are similar if they have an overlapping semantic (non-empty intersection of the representative sets). More formally, we have a lookup database of n strands $\mathbb{A} = a_1, \ldots, a_n$ and a query strand q. The lookup contains strands that are similar to q and strands that are dissimilar. Given a number k, the network has to return the k strands in $\mathbb{A}$ that are most similar to q.

5.4.2 Extrinsic Tasks

In the extrinsic tasks, BinBert will be used as the encoding layer of a neural architecture and fine-tuned end-to-end.

Strand Similarity

The **extrinsic strand similarity** is the same task as its intrinsic version; in this case, we fine-tune BinBert using a dataset of similar and dissimilar pairs of strands. We decided to include this task as it will quantify the effect of fine-tuning on the creation of semantic preserving embeddings.

Strand and Block Compiler Provenance

In the **strand compiler provenance** task, the architecture has to recognize the compiler and the optimization levels used to generate a particular strand. This task has been previously proposed on functions [106] and fragments of code [112]. In

compiler provenance, the network has to recognize the syntactic signature that a compiler, or optimization level, produces. The **block compiler provenance** task is analogous but at the block level.

Strand Recovery and Execution

We designed two novel tasks that test the semantic understanding of assembly sequences. In **strand recovery**, we provide the network a basic block of the CFG where one instruction is marked. The DNN has to recognize all instructions in the same strand of the marked instruction. This task tests the understanding of the inputs/outputs of instructions, as the network has to infer the dependency created by implicit registers such as RFLAGS.

In **strand execution**, a strand and a question are given to the network. The question is composed of an assignment for the inputs and a marked output. The network has to predict the value of the marked output. This task is interesting as it forces the network to concretely execute the snippet of assembly code and compute the correct output.

Function Level - Compiler Provenance and Similarity

Finally, our multitask benchmark includes two tasks at the function level. In the **function compiler provenance** task, the network is tasked with analyzing an entire binary function to predict the compiler used to generate the function, as well as the optimization level. In the **function similarity** task, given a database of functions and a set of query functions, the network must identify, for each query, the similar functions in the dataset. We use the standard definition of function similarity [107]: two assembly functions are considered similar if they are derived from the same source code but compiled with different compilers or optimization levels. Function similarity is currently a prominent research topic [49, 107, 121, 168, 175] due to its significant security implications, which are discussed in depth in Section 5.9.

5.5 Datasets, Pre-Training and Implementation Details

Our datasets and the implementation of our system are released and open-sourced. They can be found at https://github.com/gadiluna/BinBert.

5.5.1 Datasets

We used three datasets, a pre-train dataset PTData used to pre-train BinBert, a test dataset TestData used for the downstream tasks, and another test dataset SimTestData specifically used for the similarity tasks. Our datasets are for X64.

PTData - Pre-training Dataset

The pre-training dataset, PTData, includes the following projects: ccv-0.7, binutils-2.30, valgrind-3.13.0, libhttpd-2.0, openssl-1.1.1-pre8, openmpi-3.1.1, coreutils-8.29, gsl-2.5, gdb-8.2, postgresql-10.4, ffmpeg-4.0.2, and curl-7.61.0. These projects were compiled using compilers clang-3.8, clang-3.9, clang-4.0, clang-5.0, gcc-3.4, gcc-4.7,

gcc-4.8, gcc-4.9, and gcc-5.0. For each compiler, we compiled every project four times, once for each optimization level in $O0, O1, O2, O3$.

We used radare2 (version 5.6.0) to extract function signatures and angr (version 9.1.11611) to obtain CFGs, basic blocks, and strands, as well as to derive the set of symbolic expressions from each strand. After removing duplicates, we obtained 17.215.046 pairs in the form $(strand, simexpr)$ as dataset for the SSM task.

TestData - Test Dataset

The test dataset, TestData, is derived from projects diffutils-3.7, findutils-4.7.0, inetutils-2.0, mailutils-3.10, and wget-1.20.3. We used compilers clang-3.8, clang-6.0, clang-9, gcc-5, gcc-7, gcc-9, and icc-21, along with the four optimization levels $O0, O1, O2, O3$, to generate the raw binaries. These raw binaries will be used to create specific datasets for each task. As some operations are task-dependent, we discuss the specific split and format of the data in each task's experimental section. We have ensured the removal of duplicates so that the same sample will not be present in both fine-tune and test splits.

SimTestData - Similarity Test Dataset

The similarity test dataset, SimTestData, comprises the following projects: putty-0.74, ImageMagick-7.0.10-62, sqlite-3.34.0, gmp-6.2.0, zlib-1.2.11, nmap-7.80 and libtomcrypt-1.18.2. Similar to the previous dataset, we used compilers clang-3.8, clang-6.0, clang-9, gcc-5, gcc-7, gcc-9, and icc-21, with the four optimization levels $O0, O1, O2, O3$. This dataset has been created to construct a large benchmark for testing various state-of-the-art solutions for similarity tasks.

Model Parameters, Pre-Training and Implementation Details

Our model is built using python 3, with pytorch (version 1.10.2+cu113) [116] and hugginface (version 4.16.02) [157]. We trained it on a DGX A100, using 4 A100 GPUs.

BinBert parameters are: sequence length 512, hidden size 768, intermediate size 3072, 12 attention heads and layers. The total number of parameters is 92,645,512 . We used AdamOptimizer and learning rate 0.0001. The masking rate mp is 0.3. The batch size for each device is 32 with two steps of gradient accumulation; having 4 GPUs the equivalent batch size is 256. We trained for 1 epoch using 1425 steps of warmup. The time needed for the pre-training is 46 GPU hours.

5.6 Experimental Evaluation

In our evaluation we answer the following experimental questions:

RQ 1 Is an execution-aware transformer model trained on strands of assembly instructions and symbolic expressions the state-of-the-art assembly model for binary understanding?

RQ 2 What is the impact of pre-train on the performance across several binary understanding downstream tasks?

RQ 3 What is the impact of using a symbolic execution-aware pre-training? How it fares against the standard Masked Language Modeling (MLM) of Bert?

To answer **RQ 1** we compare BinBert with PalmTree and Trex on all the tasks of our benchmark. We also compare BinBert with state-of-the-art function similarity solutions SAFE [107], GNN-BoW, GMN-BoW [93, 105] and BinShot [4] on the extrinsic function similarity task. For a fair comparison, we retrain both PalmTree, GNN-BoW and GMN-BoW on our pre-train dataset, by using the same parameters of the original papers. On all the extrinsic tasks we fine-tune PalmTree and Trex on exactly the same dataset we used to fine-tune BinBert. It is worth noticing that PalmTree authors highlighted the possibility of fine-tuning their system but they did not explore this possibility in their paper; while Trex authors test the fine-tuning on the function similarity task only. We are interested in assessing the effective contribution of the symbolic expressions used during pre-training (**RQ 3**) and the pre-training itself on downstream applications (**RQ 2**). To do so we will use the following baselines:

- **BinBert-MLM**: This is a model pre-trained on strands only with the standard Masked Language Modeling Task (MLM). In MLM only the assembly of a strand is given to the transformer during pre-training, the pre-training task is to recover masked tokens. The gap between this model and BinBert quantifies the impact of symbolic execution-awareness.
- **BinBert-FS**: This is a transformer encoder with the application-specific neural architecture trained from scratch on the specific downstream task. The gap between this model and BinBert quantifies the impact of pre-training.

For the fine-tuned models we use the notation: **BinBert-FT**, **BinBert-MLM-FT**, **PalmTree-FT**, and **Trex-FT**.

For each task, we will provide the details of the dataset, the solution we employed to solve the problem, the metrics used to evaluate the performance, and the final results.

5.6.1 Intrinsic Tasks

The results for intrinsic tasks are reported in Table 5.2.

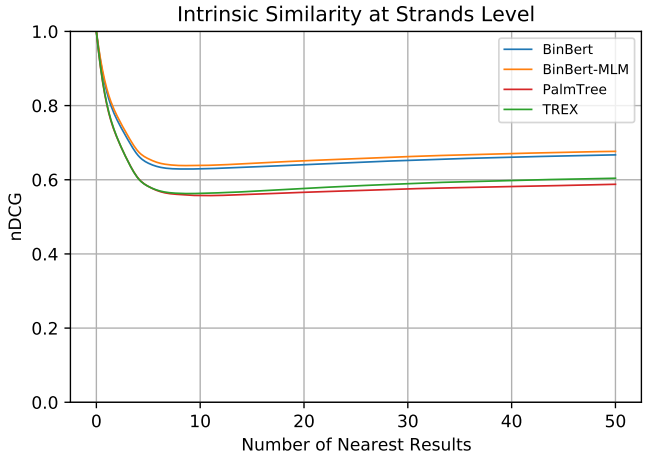
Opcode Outlier Detection

Dataset We used 43618 different instructions to create a dataset of 50000 sets of 5 instructions each. These sets are created as we have defined in Section 5.4.1. Analogously to [91], opcodes are categorized according to the *x86 Assembly Language Reference Manual* [5]. In Table 5.3 we reported the classes that we have used to categorise instructions for the opcode outlier detection. In particular, for each of the ten classes in Table 5.3 we created 5000 groups, in each group the outlier is chosen

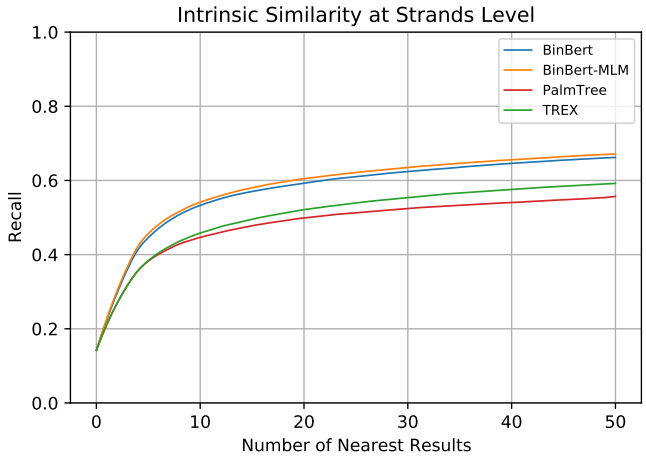
[5] https://docs.oracle.com/cd/E26502_01/html/E28388/ennbz.html

			Models											
			BinBert			**BinBert-MLM**			**PalmTree**			**Trex**		
			Accuracy			*Accuracy*			*Accuracy*			*Accuracy*		
Outlier Detection	**Opcode**		**81.9** ± 0.17			71.9 ± 0.17			77.6 ± 0.21			75.7 ± 0.18		
			Prec.	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*
Similarity	**Strands**	top-10	38.9	52.2	62.9	**39.5**	**53.0**	**63.8**	33.0	43.8	55.8	33.3	44.8	56.3
		top-20	22.9	58.9	63.9	**23.4**	**60.1**	**65.0**	19.4	49.6	56.5	19.9	51.7	57.5
		top-40	12.8	64.4	66.0	**13.0**	**65.4**	**67.0**	10.8	53.9	58.1	11.3	57.4	59.7

Table 5.2. Intrinsic evaluation results.



(a) nDCG for the top-k answers with $k \leq 50$.



(b) Recall for the top-k answers with $k \leq 50$.

Figure 5.3. Results for the **intrinsic strand similarity** task. Database of 42547 strands, average on 4942 queries.

at random among all the instructions of the other classes. This ensures that we have a balanced dataset: for each possible pair of classes on average, we have the same number of groups in our dataset.

Metrics and Results To solve this task, we embed each instruction in the set and we evaluate if the embeddings are able to distinguish the outlier; this is done by computing the distance of each embedded instruction from the others and by predicting as outlier the most distant. An instruction embedding is computed by mean pooling the instruction tokens' hidden states in the second last layer of BinBert (we take the second last layer as it is less influenced by the pre-training task). The evaluation metric is the *accuracy* of [28]:

$$Accuracy = \frac{\sum_{s \in \mathbb{S}} outlier(s)}{|\mathbb{S}|} \tag{5.5}$$

where $\mathbb{S}$ is the dataset composed of instruction sets and $outlier(s)$ is equal to 1 if the outlier in the instruction set s is detected and 0 otherwise.

We compute the mean accuracy and standard deviation on 10 runs of the experiment on different datasets (each composed of 50k sets), the results are in Table 5.2. BinBert achieves the best performances (81.9 accuracy) and it shows a great improvement over BinBert-MLM (71.9 accuracy): this confirms that symbolic expressions clearly enrich the semantic learned by the model for each instruction. It also outperforms PalmTree and Trex (77.6 and 75.7 accuracies respectively) by a wide margin. The performances of PalmTree are not dominated by a transformer trained on MLM. We believe that this is so because PalmTree has been explicitly designed to be an instruction embedding solution, while BinBert-MLM has been trained on sequences. Moreover, Trex outperforms BinBert-MLM, meaning that micro-execution traces used by Trex enhance the semantic of each instructions. However, Trex does not outperform BinBert: thus confirming that symbolic expressions are more powerful than concrete execution values used by Trex. In Section 5.8.1 we report a qualitative analysis with the clusters of opcodes learned by BinBert.

Similarity at Strand Level

Dataset We use a database $\mathbb{A}$ of 42547 strands on which we perform 4942 queries $\mathbb{Q}$ which correspond to the number of equivalence groups. That is we perform one query for each group of similar functions. On average for each query, we have 8.61 (s.d. $\pm$4.9) similar elements in $\mathbb{A}$.

Metrics and Results To solve the task we compute an embedding vector $\vec{q}$ for each query q and an embedding vector $\vec{a}$ for each strand a in the lookup database $\mathbb{A}$. This is done by averaging all the instruction tokens' hidden states in the second last layer of BinBert. For each query vector $\vec{q}$ we compute the cosine similarity with all $\vec{a} \in \mathbb{A}$; we return the ordered list of the top-k similar elements $R_q = (r_1, \ldots, r_k)$. Using R_q we compute: precision, number of true similars in R_q over k; recall, number of true similar in R_q over $\#sim(q)$, that is the number of items similar to q in $\mathbb{A}$; and nDCG. The nDCG is a measure used in information retrieval. It is defined as:

Type	Opcodes
Data Movement	mov, push, pop, cwtl, cltq, cqto,cqtd
Unary Operations	inc, dec, neg, not
Binary Operations	add, sub,imul, xor, or, and, lea, leaq
Shift Operations	sal, sar, shr, shl
Comparison and Test Intructions	cmp, test
Conditional Set Instructions	sete, setz, setne, setnz, sets,setns, setg, setnle,setge, setnl, setl, setnge,setle, setng, seta, setnbe, setae, setnb, setbe, setna
Jump Instructions	jmp, je, jz, jne, jnz, js, jns, jg, jnle,jge, jnl, jl, jnge, jle, jng, ja, jnbe, jae,jnb, jb, jnae, jbe, jna
Conditional Move Instructions	cmove, cmovz, cmovne, cmovenz, cmovs, cmovns, cmovg, cmovnle, cmovge, cmovnl, cmovnge, cmovle, cmovng, cmova, cmovnbe, cmovae, cmovnb, cmovb, cmovnae, cmovbe, cmovna
Procedure Call Instructions	call, leave, ret, retn
Floating Point Arithmetic	fabs, fadd, faddp, fchs, fdiv, fdivp, fdivr,fdivrp, fiadd, fidivr,fimul, fisub, fisubr, fmul,fmulp, fprem, fpreml, frndint, fscale, fsqrt, fsub, fsubp, fsubr, fsubrp, fxtract

Table 5.3. Opcode classes used to categorise instructions for the opcode outlier detection task.

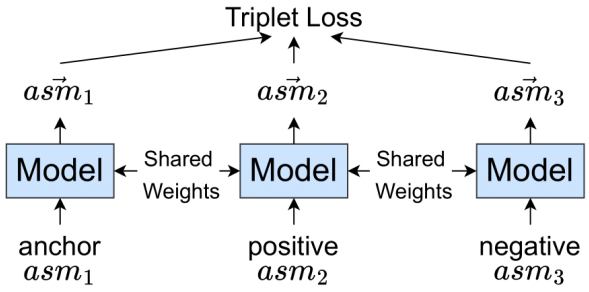


Figure 5.4. Scheme for the triplet loss: asm_i can be either a strand or a function and Model is either BinBert, BinBert-ML.

$$nDCG = \frac{\sum_{i=1}^{k} \frac{sim(r_i,q)}{log(1+i)}}{\sum_{i=1}^{\#sim(q)} \frac{1}{log(1+i)}} \tag{5.6}$$

Where $sim(r_i, q)$ is 1 if q is similar to r_i and 0 otherwise. The quantity at the denominator is the scoring of a perfect answer, and the number at the numerator is the scoring of our system. The nDCG is between 0 and 1, and it takes into account the ordering of the items in R_q, giving better scores when similar items are ordered first. As an example let us suppose we have two results for the same query: $(1, 1, 0, 0)$ and $(0, 0, 1, 1)$ (where 1 means that the corresponding index in the result list is occupied by a similar item and 0 otherwise). These results have the same precision (i.e., $\frac{1}{2}$), but nDCG scores the first better. We average the per-query precision, recall, and nDCG to obtain the final metrics. Results are shown in Table 5.2 and in Figure 5.3. We can see that BinBert and BinBert-MLM achieve the best performance on precision, recall and nDCG, the MLM model has a slight edge on BinBert.

5.6.2 Extrinsic Tasks at Strand and Basic Block Level

Results for extrinsic tasks are shown in Table 5.4.

Strands Similarity

Dataset We fine-tuned BinBert on the task of recognizing whether strands are similar or dissimilar. We created a dataset of 49974 samples, each being a triplet consisting of an anchor strand a, a positive strand p (similar to the anchor), and a negative strand n (dissimilar from the anchor). We split it into train and validation, resulting in 39978 strands triplets for the training set and 9996 for the validation. We test the fine-tuned models on the same test sets used in the corresponding intrinsic tasks (see Sec.5.6.1).

Fine-tuning We fine-tune the model using a siamese architecture [27, 107, 160] with the triplet objective function [130] (see Figure 5.4). In this architecture three instances of the embedding network are used, each instance produces the embedding of the corresponding entity in a triplet. The resulting embeddings are used during

training to minimize the triplet margin loss that has the following mathematical form:

$$\mathcal{L}(\vec{a}, \vec{p}, \vec{n}) = \max\{\|\vec{a} - \vec{p}\|_2 - \|\vec{a} - \vec{n}\|_2 + \epsilon, 0\} \tag{5.7}$$

The training process instructs the embedding network to produce embeddings such that the euclidean distance between $\vec{a}$ and $\vec{p}$ is smaller than the euclidean distance between $\vec{a}$ and $\vec{n}$ by at least a margin of ϵ. For BinBert-FT and BinBert-MLM-FT the embeddings for the triplet loss are given by the average of all tokens of the last layer (excluded the padding).

Differently from BinBert, which can directly embed any arbitrary sequence of assembly code, Palmtree is an instruction embedding model, thus requiring an additional architecture to transform it into a model that embeds sequences. In particular, we use a bidirectional LSTM, where each cell takes as input the instruction embedding produced by PalmTree. Finally, we compute an embedding by averaging all the hidden states of the LSTM. We stress that in case of BinBert we do not need to use an LSTM to compute a function embedding but we use the average of the tokens of the last layers.

Regarding Trex, we used the same finetuning process used in the original paper for the function similarity test.

We fine-tune each model for 20 epochs, selecting the epoch with the best AUC on validation.

Metrics and Results Results for strands similarities are in Table 5.4. In Figures 5.5 there are the results of strand similarity for $k \in [0, 50]$.

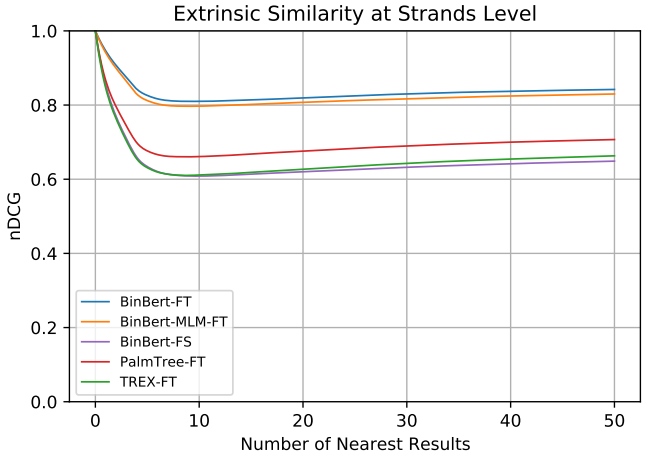
Also in this case BinBert achieves the best performances. The gap between BinBert-FT and the other models is much wider than in the intrinsic case. A possibility is that BinBert learns a wider semantic during pre-training, solving more efficiently the similarity task after fine-tuning. This explains the gap between BinBert-FT and BinBert-MLM-FT.

The great impact of pre-training can be appreciated by looking at the performance of BinBert-FT and BinBert-FS. BinBert-FS has been only trained on the fine-tune dataset so it cannot leverage a learned semantic, the fine-tune dataset has not enough data to make up for this disadvantage and to train a big transformer model.

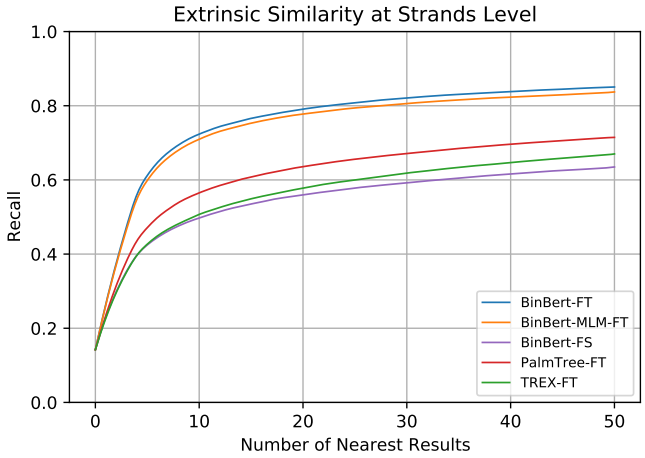
Compiler Provenance

As in other works on compiler provenance [106], we train and test the networks on the task of *Compiler Classification*, that is detecting the compiler family that has generated a sample, and *Optimization Classification*, detecting the optimization level used to generate a sample.

Dataset The compiler provenance dataset is made up of samples in which, depending on the granularity, a strand or a basic block is associated with two labels: the compiler family with its corresponding version and the optimization used. The strand compiler provenance dataset has 30760 samples, split into 24564 samples for the training set, 3104 samples for the validation set, and 3092 samples for the test sets. Each compiler family contains 4394.29 samples on average (s.d. $\pm$185.08), thus



(a) nDCG for the top-k answers with $k \leq 50$.



(b) Recall for the top-k answers with $k \leq 50$.

Figure 5.5. Results for the **extrinsic strand similarity** task. Database of 42547 strands, average on 4942 queries.

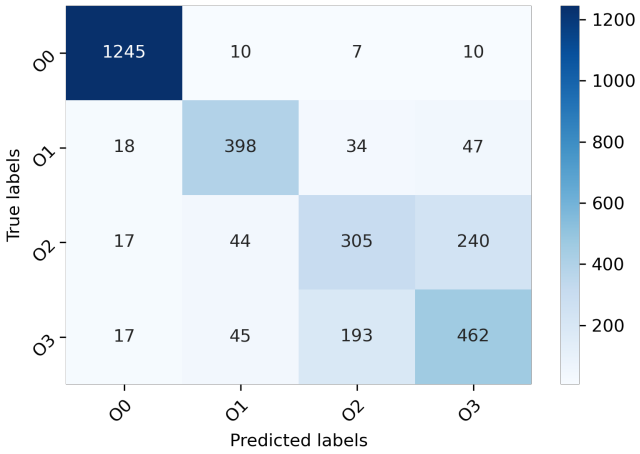
resulting in a balanced dataset. In terms of optimization levels, there is an imbalance caused by O0 optimization, which has a higher number of strands, while other optimization levels (O1, O2, O3) are more balanced. Specifically, O0 optimization level contains 12727 samples, while other levels contain 6011 samples on average (s.d. ±775.01). The basic blocks compiler provenance dataset has 73252 samples, split into 58625 training samples, 7258 validation samples, and 7369 test samples. Each compiler family contains 10464.57 samples on average (s.d. ±775.01), thus resulting in a balanced dataset. Regarding optimization levels, there exists an imbalance similar to the strand case. In fact, the O0 optimization level contains 26380 samples, while other levels contain 15624 samples (s.d. ±1828.63).

Fine-tuning We fine-tune BinBert, BinBert-MLM, and Trex on both compiler and optimization classification by adding a linear layer followed by the softmax function on top of the last layer hidden state corresponding to the `[CLS]` token. For PalmTree we use an LSTM over the instruction embedding tokens generated; to obtain a classifier we attach a linear layer with the softmax on top of the last hidden states of the RNN. We fine-tune each model for 20 epochs, selecting the epoch with the best classification accuracy on validation.

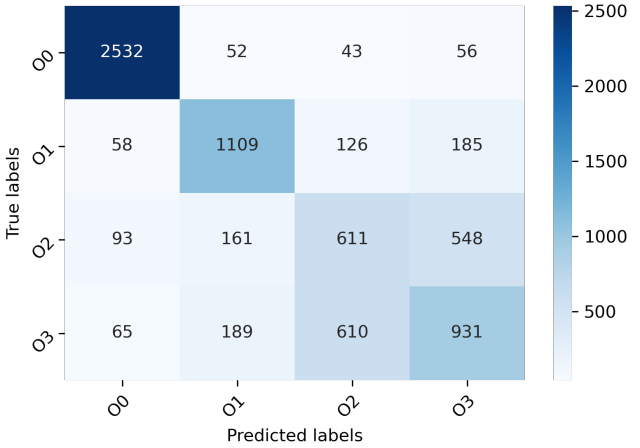
Metrics and Results To assess the performance of the models, we utilized macro precision, recall, and F1 score. These metrics guarantee equal treatment of all classes, thereby mitigating the potential for misleading outcomes in the presence of imbalances as discussed earlier. Results for both compiler and optimization classification are shown in Table 5.4. In this case, BinBert-FT is slightly worse than BinBert-MLM-FT, we believe that this is due to the fact that recognising a compiler signature is a syntactic task. PalmTree-FT and Trex-FT have the worse performance among all fine-tuned models. Again we can see that pre-training is important as the from-scratch model performs consistently worse than its fine-tuned counterpart. Results are similar for the optimization classification task. Since the optimization dataset has some imbalances, we reported in Figure 5.6 the confusion matrices obtained by using BinBert on the optimization classification tasks for both strands and basic blocks. We can observe that BinBert is better at distinguishing optimized (O1, O2, O3) vs unoptimized code (O0) rather than recognizing the specific optimization level used to compile it.

Strand Recovery

Dataset The dataset is made of 9265 basic blocks, each block contains at least 5 disjoint strands (i.e. the strands do not overlap on instructions). We split the dataset into 7411 training samples and 927 samples for both validation and test. We model strand recovery as instruction classification; given the instructions of a basic block and the final instruction of one of its strands, we aim at classifying the other instructions as either belonging to the same strand as the marked instruction or not. Thus, the classification of each instruction is a binary classification task. To mark an instruction we surround it with a special token. The total number of instructions is 105288, 13230, and 13138 for the training, validation, and test set respectively. The total number of instructions belonging to the positive class is 29% in each set.



(a) Confusion matrix for the optimization classification task at strand level with BinBert.



(b) Confusion matrix for the optimization classification task at basic block level with BinBert.

Figure 5.6. Results for the **extrinsic compiler provenance** task at strand and basic block level.

Fine-tuning We fine-tune BinBert, BinBert-MLM, and Trex by attaching a classification head on top of the last layer hidden states of the first token of each instruction; the network will output 1 if an instruction is part of the strand to be recovered and 0 otherwise. As for previous tasks, we use an LSTM on top of PalmTree and we put a classification head on the hidden states of the first token of each instruction. We fine-tune each model for 20 epochs, selecting the epoch with the best classification accuracy on validation.

Metrics and Results To evaluate the performance of the models, we used precision, recall, and F1-score of the positive class, which is the minority class as well. Results are shown in Table 5.4. We can see that the best-performing model is BinBert-FT, it is outperformed only by Trex-FT in terms of precision, but it markedly surpasses PalmTree-FT on all the metrics considered. BinBert-MLM-FT achieves the same precision as BinBert-FT but smaller recall. We believe the reason to be the execution-awareness of BinBert-FT that allows the model to recover more instructions in a strand. For the sake of completeness, we reported the performance of BinBert on the negative class as well: 99.6 Precision, 99.0 Recall and 99.3 F1 score. A qualitative analysis showing the change of internal attention weights of BinBert during fine-tuning is in Section 5.8.2.

Tasks			Models														
			BinBert-FT			BinBert-MLM-FT			BinBert-FS			PalmTree-FT			Trex-FT		
			Prec.	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*	*Prec.*	*Rec.*	*nDCG*
Similarity	Strands	*top-10*	**54.3**	**71.1**	**81.0**	53.1	69.7	79.7	36.8	48.8	60.9	41.4	55.3	66.0	37.0	49.5	61.0
		top-20	**31.7**	**78.6**	**81.8**	31.1	77.4	80.6	21.8	55.6	61.9	24.6	63.1	67.5	22.0	57.3	62.5
		top-40	**17.3**	**83.7**	**83.7**	16.9	82.2	82.4	12.3	61.4	64.1	13.9	69.4	69.9	12.8	64.4	65.3
	Functions	*top-5*	**78.5**	**38.9**	**83.4**	75.9	37.4	81.3	39.5	19.3	51.4	60.5	29.1	69.4	71.9	35.2	78.1
		top-10	**59.2**	**55.3**	**74.7**	56.2	52.3	71.8	23.9	22.8	39.9	39.9	36.8	56.7	51.9	48.3	67.8
		top-25	**31.5**	**69.6**	**74.0**	29.8	65.8	70.8	12.0	28.0	38.0	20.3	45.2	54.2	27.3	60.6	65.5
			Prec.	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*
Compiler Classification	Strands		73.7	73.6	73.6	**75.4**	**75.2**	**75.2**	45.8	45.9	44.5	69.4	68.8	68.4	69.5	68.9	68.7
	Basic Blocks		72.6	72.3	**72.4**	**72.7**	**72.4**	**72.4**	54.2	53.7	53.6	69.9	68.8	69.0	70.0	68.2	68.6
	Functions		**89.0**	87.6	88.2	88.9	**88.7**	**88.8**	74.3	69.6	74.7	86.0	84.0	84.7	88.5	86.1	86.9
Optimization Classification	Strands		**73.4**	73.2	73.2	**73.4**	**73.5**	**73.4**	58.4	59.0	57.1	69.8	67.2	68.2	69.9	69.8	69.6
	Basic Blocks		65.9	66.1	66.0	**66.2**	**66.4**	**66.3**	57.6	57.8	53.7	63.1	63.1	67.2	60.4	61.0	60.3
	Functions		**70.0**	**70.0**	**70.0**	68.8	68.4	68.4	62.0	59.4	58.6	66.7	66.2	66.3	67.7	67.5	67.5
Strand Execution			**80.1**	**80.0**	**79.1**	78.4	77.0	76.6	17.2	17.7	16.6	25.1	23.8	23.3	18.4	20.1	16.7
Strand Recovery			96.9	**98.7**	**97.8**	96.9	98.4	97.6	65.7	29.9	41.1	81.1	75.5	79.2	**98.0**	97.0	97.5

Table 5.4. Extrinsic evaluation results.

Strand Execution

Dataset We created a dataset for the strand execution task by taking strand-symbolic execution pairs, assigning concrete values to input variables, and evaluating its concrete output. In particular, we randomly assign values between 0 and 100 to input variables and we take only strands whose output is not greater than 200. Our dataset only contains strands computing the value of a register or a predicate of a conditional branch. The dataset is composed of 40000 training samples, and 5000 validation and test samples (total 50k). In each dataset, approximately 83% of the samples have an output value below 100. For each output value between 0 and 100 there is an average number of 41 strands (s.d. $\pm$7.65).

Each sample is made up of strand instructions followed by concrete assignments of input variables and the query output variables (only in the case of register outputs); the label for such a sample is the concrete value of the output variable. For instance, consider the strands `mov eax, dword ptr [rbp - 180] sub eax, 1` and the value 9 assigned to `dword ptr [rbp - 180]`. The corresponding sample will be `mov eax, dword ptr [rbp - 180] sub eax, 1 [SEP] dword ptr [rbp - 180] = 9 [SEP] rax`. The network has to predict the value for register `rax`, in this case 8.

Fine-tuning We model this problem as a sequence classification task, thus we used the same architectures used for the compiler provenance tasks (see Section 5.6.2). Regarding Trex, since it uses a bi-LSTM to deal with concrete values, we pass our concrete input inside that LSTM architecture.

Metrics and Results To evaluate the performance of the models and to address dataset imbalances we used macro precision, recall and F1 measures. We reported results obtained by all the models in Table 5.4. BinBert-FT achieves the best performances. PalmTree-FT and Trex perform poorly on this task. In the case of Palmtree we believe that this is due to the fact that it is a pure instruction embedding model so even when fine-tuned it cannot transfer to the upward neural architecture A sequence-related knowledge. On the contrary, both BinBert-FT and BinBert-MLM-FT are trained on sequences; however, BinBert-FT has the edge thanks to its execution-awareness. Regarding Trex, we believe that low performances are due to its architecture that treats concrete values in a separate network. In this case, the pre-training has a great impact as we can see by the poor performance of BinBert-FS.

5.6.3 Extrinsic Tasks at Function Level

To address function-level tasks, we employ BinBert on the Linearized CFG. This approach serves two purposes. Firstly, it evaluates whether the drawbacks of such a representation, as discussed in Section 5.2.1, are offset by the pre-training phase. During this phase, the noise from unrelated instructions is filtered out through the use of strands. Secondly, it aims to assess the representational capacity of our assembly code model independently of complex architectures or elaborate function representation. This is crucial for determining that the source of performance improvements stem from the assembly code model itself and are not due to other structures, either integrated with or built upon our model, which could confound the evaluation of its true effectiveness.

Similarity

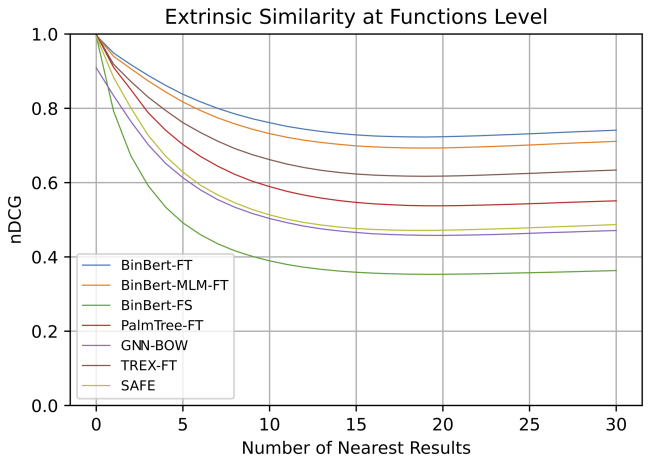
Fine-tune Process and Dataset The function similarity dataset has exactly the same format as the similarity tasks at strand level (see Section 5.6.2). In particular, the training dataset contains 18834 function triplets, while the validation set contains 7650 triplets.

On top of the embedding models, we use the same architectures that we employed to solve the other similarity tasks (see Section 5.6.2). Specifically, for PalmTree we use it on the linearized CFG using an LSTM on top of the instruction embeddings.

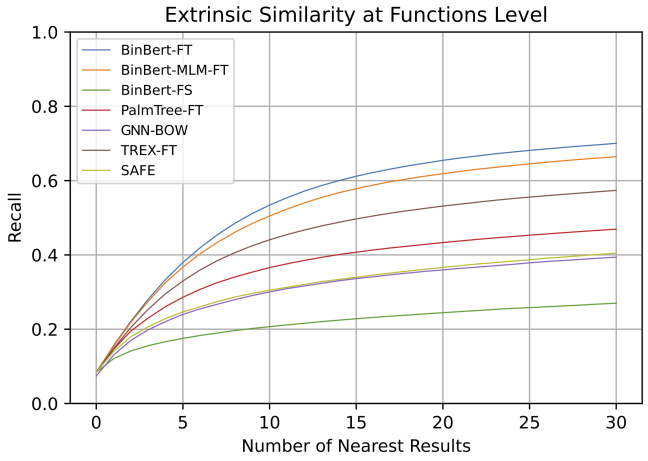
We compare our solution with some other approaches that have been specifically designed to tackle the binary similarity problem. These include SAFE [107], Trex [121], GNN-BoW, GMN-BoW [93,105] and BinShot [4]. SAFE uses word2vec [144] to create instruction embeddings that are then fed to a self-attentive neural network [98] for learning the final embeddings for functions. Binshot utilizes a transformer encoder that has been pre-trained on the MLM task and is subsequently fine-tuned using pairs of similar and dissimilar functions. BinShot calculates a weighted distance vector from a given pair of functions and then employs a fully-connected layer on this vector to output a final similarity score. GNN-BoW and GMN-BoW are both based on graph neural networks and are the best-performing solutions for the binary similarity problem according to [105]. In these approaches, the function control flow graph (CFG) is represented as a graph, and the nodes are represented by the bag of words (BoW) of the 200 most frequent opcodes contained in each block. While GNN-BoW computes the embedding of each function independently and then computes the similarity score, GMN-BoW computes the embeddings of a pair of graphs simultaneously. However, even though GMN-BoW achieves the best results, it cannot be used in some scenarios due to its time performance. This is because, unlike traditional embedding models, the function embeddings cannot be precomputed since they depend on the query function. This makes large-scale testing infeasible. Similar observations can be done for BinShot, in which the model is designed to produce a similarity score between function pairs rather than outputting function embeddings. For this reason, we performed two experiments on different datasets. The first experiment was conducted on a **larger dataset**, where all models were evaluated except GMN-BoW and BinShot. This dataset comprised 58773 functions, with 5000 of them serving as queries (the number of queries corresponds to the number of equivalence groups). On average, each query had 11.7 similar entities (s.d. $\pm$5.29). The second experiment was conducted on a **reduced dataset**, where GMN-BoW and BinShot took a reasonable amount of time to compute. This dataset consisted of 5962 functions, with 422 of them serving as queries. On average, each query had 15 similar entities (s.d. $\pm$5.91).

Results Results are in Figure 5.7 and Table 5.4. BinBert achieves the best performances, beating also GNN-BoW, SAFE and Trex which are specific for the function similarity problem. PalmTree performs worse than BinBert, confirming our initial intuition about the drawbacks of the lack of context and the isolated instruction embeddings. SAFE has lower performances than PalmTree. This suggests that even if they are both based on fixed instructions embeddings, a transformer encoder is more capable of capturing an instruction semantic than word2vec. Unsurprisingly, the Trex model beats PalmTree. Also in this task, BinBert-FS is below its corresponding fine-tuned version highlighting the importance of pre-training. The impact of execution awareness can be seen in the advantage of BinBert-FT on BinBert-MLM-FT. Finally, the results for the experiments on the reduced dataset are in Figure 5.8, also in this case we can see that BinBert-FT is the state-of-the-art and it performs better than

GMN-BoW and BinShot.



(a) nDCG for the top-k answers with $k \leq 30$.



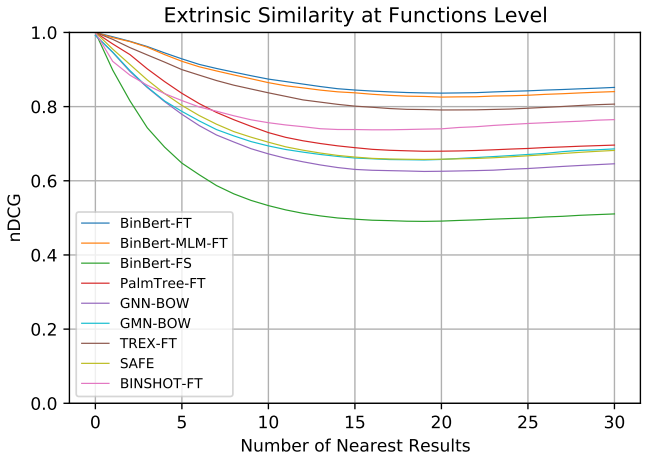
(b) Recall for the top-k answers with $k \leq 30$.

Figure 5.7. Results for the **function similarity** task on the largest dataset. Database of 58773 functions, average on 5000 queries.

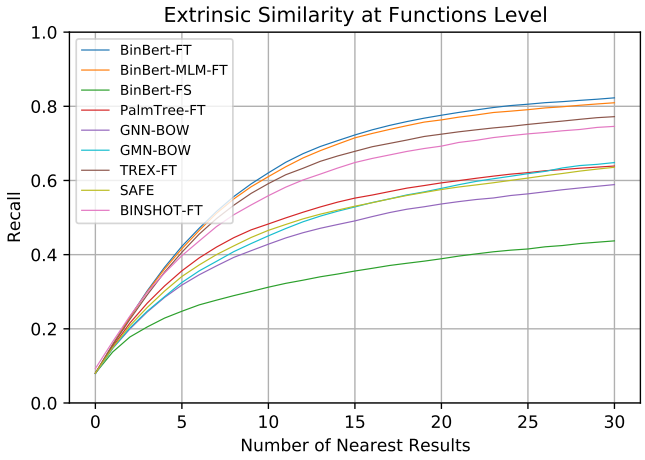
Compiler Provenance

Fine-tune Process and Dataset The compiler provenance dataset has the same structure as the strand and block compiler provenance tasks. It contains 39017 functions in the training set and 4877 functions in the validation and test set. Each compiler family contains 6967.42 samples on average, thus resulting in a balanced dataset (s.d. ±1600.42). Regarding optimization levels, there exists an imbalance similar to the strand and basic block cases. In fact, the O0 optimization level contains 16258 samples, while other levels contain 10838 samples (s.d. ±2589.33).

We used the same architectures that we employed to solve the other compiler provenance tasks.



(a) nDCG for the top-k answers with $k \leq 30$.



(b) Recall for the top-k answers with $k \leq 30$.

Figure 5.8. Results for the **function similarity** task on the reduced dataset. Database of 5962 functions, average on 422 queries.

Results We reported the macro precision, recall, and F1 score for different models in Table 5.4. Results are similar to the compiler provenance task at strand and block level, thus the same considerations hold. In Figure 5.9 we reported the confusion matrices obtained by using BinBert on the compiler and optimization classification tasks. We can observe that BinBert can clearly distinguish among different compiler families and it only gets confused with different versions in the same family. Similar behavior can be observed in the optimization classification task. It is easier for BinBert to distinguish optimized (O1, O2, O3) vs unoptimized code (O0) than to recognize the specific optimization level used to compile it.

Results from the experimental evaluation confirm that BinBert is the current state-of-the art for assembly code models. It shows improvement over PalmTree, Trex, and other deep learning solutions specifically tailored for a certain task. Execution awareness has a marked impact on semantic tasks. Interestingly, on some syntactic tasks, execution awareness does not increase the performance of the model. Our evaluation highlights the great impact of pre-training on downstream tasks with small-size datasets.

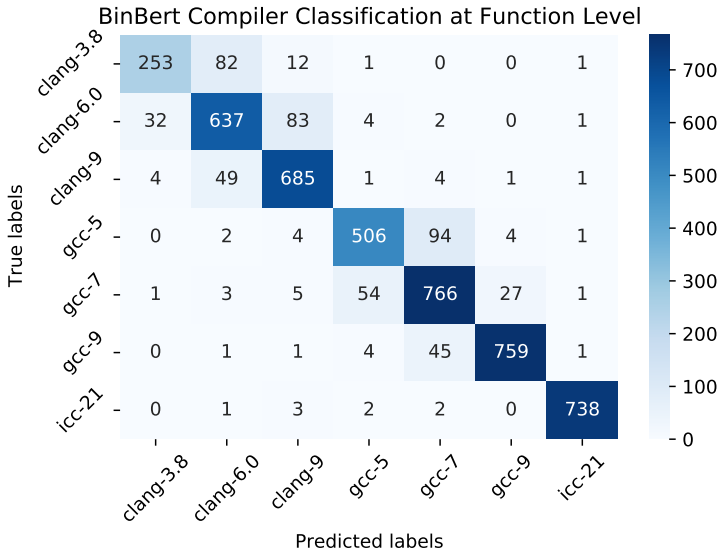
5.6.4 Tokenizers

In all of the aforementioned experiments, we utilized the Wordpiece tokenizer, which is a standard choice for Bert-like models. However, we also assessed the performance of the model using two additional tokenization strategies: whitespace tokenizer and unigram tokenizer. The whitespace tokenizer, employed by PalmTree and Trex, involves splitting tokens based on whitespaces and symbols. The unigram tokenizer initializes its vocabulary with a large number of symbols and progressively trims it down to obtain a smaller vocabulary. Our experimental evaluation shows that the whitespace tokenizer results in an average performance decrease of 1.88%, while the unigram tokenizer leads to a performance decrease of 8.91%.

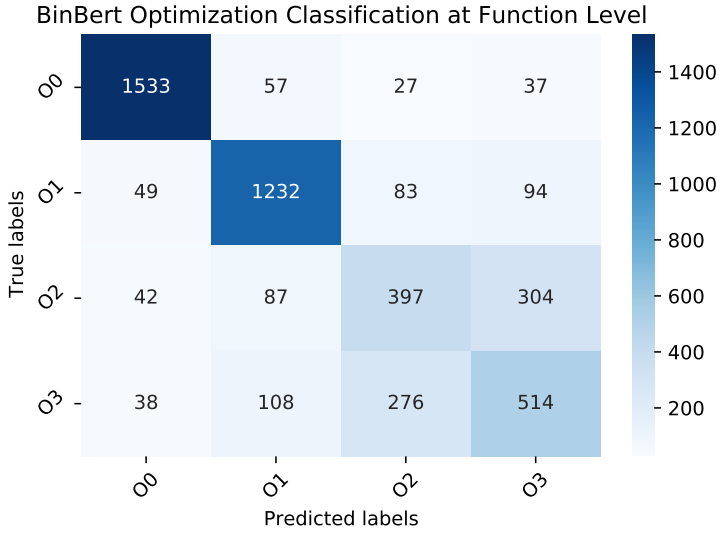
5.7 Time Performance Comparison

Table 5.5 shows the time performance of different solutions for binary similarity. The measures, in seconds, are the time required for the models to generate embeddings [6] and find the top k most similar functions in the reduced dataset. The GNN-BoW model is the fastest, likely due to its smaller size of basic block embedding. Palmtree is slower than BinBert and SAFE, possibly due to the need of applying a transformer model separately to each assembly instruction. Trex is slower than Palmtree due to its use of an additional LSTM architecture to embed numerical values. GMN-BoW is the slowest model, as it must generate a similarity score for each pair of functions in the dataset. The performance of GMN-BoW may not be practical in a real-world scenario, especially when the network has to be used to retrieve similar functions in a large database of samples. BinShot outperforms GMN-BoW in terms of speed, but it lags behind other embedding models due to the necessity of computing similarity scores for each function pair.

[6] The measures are taken after the binary function is converted in the format required by the model.



(a) Confusion matrix for the compiler classification task at function level with BinBert.



(b) Confusion matrix for the optimization classification task at function level with BinBert.

Figure 5.9. Results for the **extrinsic compiler provenance** task.

Model	Time (s)
BinBert	37.5
BinBert-MLM	37.3
Palmtree	49.7
Trex	87.9
SAFE	37.6
GNN-BoW	19.1
GMN-BoW	11765.8
BinShot	2327.9

Table 5.5. Results for time analysis.

5.8 Qualitative Analysis of Binbert

We performed a qualitative analysis of the Binbert model by examining the clusters of opcodes created by the model and by visualising its attention mechanism.

5.8.1 Opcode Clustering

We performed a qualitative analysis based on the visualization of the clusters of opcodes learned by BinBert. To do so, we first used BinBert to convert each opcode into a vector, and we then applied t-SNE [147] to visualize opcodes in a two-dimensional space. Results are shown in Figure 5.10. Opcodes are well clustered according to their semantics. Specifically, we can identify two symmetric regions in which jumps, conditional move, and conditional set are split based on the condition checked (either positive or negative). We can also identify a region containing operations performing multiplications (`imul`, `lea`) and divisions (`divq`), and other arithmetic operations (`add`, `sub`). Another interesting example is given by the region containing `ret`, `pop`, `push` and `call`: they all manipulate the stack.

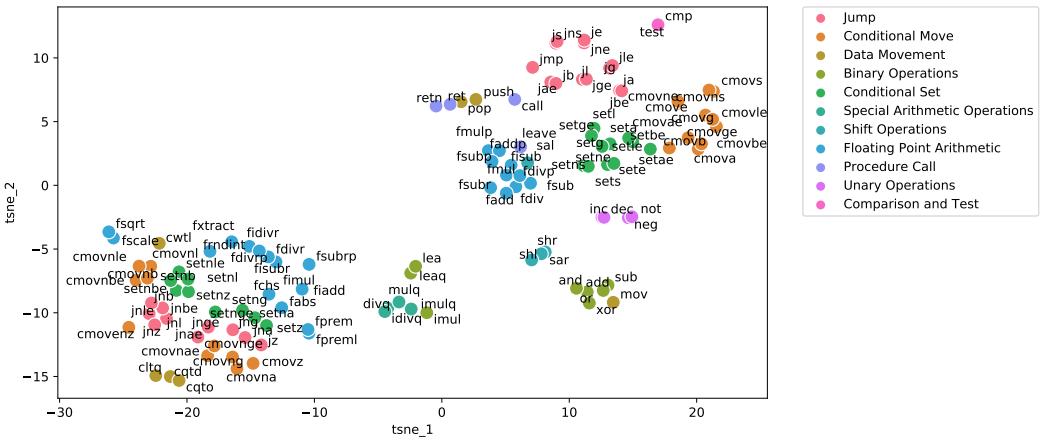
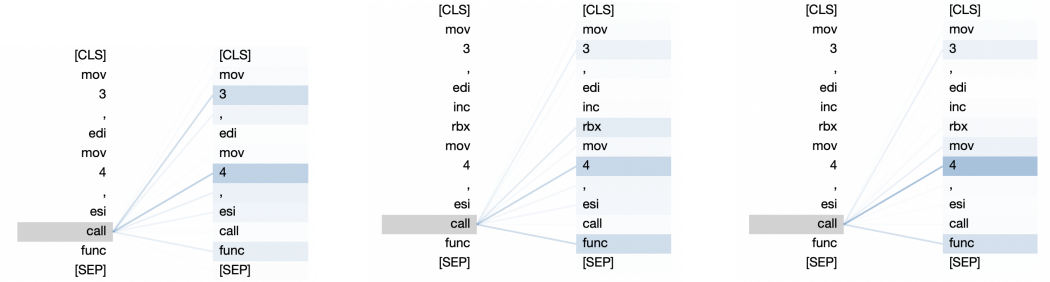


Figure 5.10. Visualization of opcode embeddings in a two-dimensional space with t-SNE.



(a) Attention weights of the last head for the embedding of the `call` token in the last layer when BinBert is fed with a strand. We can see that the network pays attention to the function arguments and the name of the called function. This shows that the network has learnt non trivial relantioshpis between the call token and the rest of the sequence.

(b) Attention weights of the last head for the embedding of the `call` token in the last layer when BinBert is fed with a non-strand. We can see that the network pays also some attention to the extraneous register `rbx`

(c) Attention weights of the last head for the embedding of the `call` token in the last layer when BinBert-FT fine-tuned on strand recovery is fed with a non-strand. We can see that the network learns that `rbx` is extraneous and do not pay attention to it.

Figure 5.11. Series of Attention Visualization snapshots, we can see that the network changes its attention paradigm during fine-tuning to more effectively tune down the noise inserted by extraneous instructions.

5.8.2 BinBert Attention Visualization

BinBert is based on a stacked attention mechanism, in each layer of the transformer, there is an attention mechanism that creates the embedding of a token by focusing on the most interesting part of the input layer. We used BertViz [152] to visualize the attention weights produced by BinBert at certain layers and heads. Specifically, we were interested in evaluating two aspects: the robustness of the attention when BinBert is fed with assembly sequences that are not strands and the effect of finetuning on those weights. We fed BinBert with a strand (Figure 5.11a). We observe the last attention head in the last layer, and we discover that when computing the embedding of the call instruction, BinBert focuses on the parameters of the call which are `3` and `4` (Figure 5.11a) and on the name of the called, meaning that it has understood the strand relationship and it understood the calling convention. We then added an instruction (`inc rbx`) to the previous strand and we gave it as input to BinBert. We can see, that the attention is still focusing on the function parameters, but it is also considering the rbx register of the outlier instruction. We then feed the same non-strand instruction to BinBert-FT finetuned on the strand recovery task. We observe that finetuning has lowered the attention weight associated with the rbx register, thus reducing the noise. This last observation shows that during

fine-tuning BinBert is able to reconfigure its attention mechanism according to the downstream task to be solved.

5.9 Security Applications of an Assembly Code Model

The BinBert assembly code model is suitable for various security applications. The model transforms sequences of assembly instructions into representations that other machine learning models or neural architectures can directly use and can be seamlessly integrated into existing solutions to leverage the semantic knowledge acquired during pre-training. In this section we show possible venue of applications of our model.

5.9.1 Reverse Engineering

An assembly code model can assist and automate multiple tasks beneficial for reverse engineers. In this scenario a fundamental challenge is understanding the semantics of unknown functions. A valuable tool is a high-level representation in the form of decompiled code. To produce decompiled code from a low-level programming language, some deep learning-based solutions use encoder-decoder architectures [29], where BinBert could replace the encoder part. Additionally, a code model can automate the task of naming assembly functions with semantically representative names. Recent studies [56, 76, 82] suggest neural architectures for this challenge, where many process sequences of assembly instructions using Transformers or RNNs as encoders. BinBert can be directly integrated into such solutions either as an encoder (for example, in [76, 82]) or as a method for creating feature vectors from assembly instructions (for example, in [53, 117]). Another area where an assembly code model proves useful is binary authorship detection, aiming to identify the author of an executable and, in malware cases, the Advanced Persistent Threat (APT) group responsible for the attack [137]. The compilation and optimization classification tasks are also significant from a security perspective. It has been recognized that specific versions of compilers and optimization levels may inadvertently introduce security vulnerabilities into cryptographic code [48]. An assembly model can furnish analysts with insights into the probable optimization level utilized for compiling a binary. This information could steer their analysis towards identifying particular security-sensitive bugs.

5.9.2 Binary Similarity

The Binary Similarity problem is a fundamental problem that has application in a wide range of security related tasks [105]. As example it has been used to classify unknown malware in families [107], or to find known vulnerabilities (CVE) in specific binaries used in production environments [4, 121, 160]. This aspect is especially important in contexts with vulnerable firmware, where a flaw at the source code level can affect numerous devices across different architectures.

5.10 Related Works

Binary code representation techniques can be subdivided into two main branches: manual features selection [10,53,95,113,117,160] and unsupervised features extraction. Since our work proposes an assembly model we discuss works that use or propose instruction embedding models. We first focus on the instruction embedding models proposed in the literature, categorising them according to the defining properties identified in the Background Section 5.2: the distributed representation learning used, the preprocessing of assembly instructions, and the extraction methodology for assembly sequences. Finally, we discuss how these models have been used to solve binary analysis tasks.

5.10.1 Distributed Representation Learning

The distributed representation learning technique is the neural architecture and the training tasks used by the instruction models to create useful embedding vectors. The most common is word2vec [144]. Word2vec has been used, with minimal modifications, by Eklavya [38], SAFE [107] and others [49, 106, 175]. A notable difference is Asm2vec [46] which uses PV-DM [86], a variation of word2vec that simultaneously creates instructions and function embeddings. We remark that PV-DM cannot be fine-tuned. PalmTree [91], Trex, Stateformer and Binshot, that we described in Section 5.2.1, use a transformer architecture, such architecture contain an embedding layer that automatically learn a distributed representation of the input tokens.

5.10.2 Preprocessing of Assembly Instructions

The preprocessing is characterised by the substitution policy for information in the raw assembly instruction and the tokenization policy. We classify the substitution policies in *aggressive* or *light*. In an aggressive policy, lots of information contained in the assembly instructions are removed or changed. An example is DeepBinDiff [49] that replaces all constants and pointers with special tokens and renames registers according to their lengths in bytes (e.g. `ecx` becomes `reg4`). InnerEye [175], instead, replaces all constants, strings, and function names with special symbols. BinDeep [7] substitutes operands based on predefined categories (e.g. general register, direct memory reference, etc.). All the above policies are aggressive; indeed, InnerEye [175] wastes fundamental information that a library function call could bring and DeepBinDiff [49] loses register names that could be relevant, for instance, to understand the data flow.

Light preprocessing policies are applied by SAFE [107] and PalmTree [91] which keep small constant values and replace values above a predefined threshold with a special token. SAFE [107] replaces all memory addresses with the same token, PalmTree uses different tokens for generic memory locations and the ones pointing to strings.

Regarding tokenization, some works consider an entire instruction as a token [107] or split an instruction into opcode and operands [46], while others use a word-based tokenizer by splitting instructions on symbols (e.g. spaces or special characters) [121],

[91]. The latter is a fine-grained tokenization strategy that is useful to reduce the vocabulary size and allow the upstream network to separately learn the semantics of each token (mnemonics, registers, etc.). No one used automatic tokenization strategies like WordPiece [158].

5.10.3 Binary Analysis Solutions using Embedding Models

Deep-learning-based solutions can be categorized into approaches using Graph Neural Networks (GNN), Recurrent Neural Networks (RNN), and Transformer architecture. [41, 106] use a GNN applied to function control flow graphs after transforming each block into a vector representation; this transformation is done by aggregating the instruction embeddings of each block. The resulting architecture is applied to the binary similarity [106], the compiler provenance problem [106], function naming problem [41]. Eklavya [38] and InnerEye [175] are examples of RNN-based solutions applied to the recovery of arguments used by a binary function [38], and basic-block similarity [175]. Another work is SAFE [107] which added a self-attention layer on top of an LSTM to solve the binary similarity problem. Among transformer-based solutions, we can find [11], [41] that apply transformers encoder-decoder architecture to recover function names from stripped binaries. Other works [4, 121, 168] use transformer architecture and specifically takle the binary similarity problem. In particular, [168] uses a transformer encoder to obtain basic blocks embeddings to be fed into a GNN which is trained in conjunction with a Convolutional Neural Network (CNN) to produce a function embedding for binary similarity. Unfortunately, [168] does not release the code and the paper lacks of the details needed to reimplement their proposal.

5.11 Conclusion

We presented BinBert, an execution-aware assembly language model. BinBert is trained on a big dataset of assembly strands and symbolic expressions. BinBert has shown state-of-the-art performance highlighting the relevance of execution-awareness. Our evaluation shows that BinBert is an encoder model that can be used to solve several tasks related to binary analysis using a fine-tune dataset of relatively small size. The generality of the model is an important strength and we believe that BinBert can be fruitfully applied to other downstream tasks as encoder layer of complex neural networks.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis explores the applicability of deep learning techniques in the context of binary analysis. In fact, the last few years have witnessed a proliferation of solutions using these techniques to solve multiple binary analysis tasks. Unfortunately, little effort has been directed toward systematizing such solutions. Consequently, the current landscape of deep neural networks (DNNs) applied to binary analysis tasks is fragmented and scattered. For this reason, this thesis makes a step forward in the research by providing an in-depth analysis of 54 research papers in the field (Chapter 3). Specifically, we identify a common deep learning pipeline and analyze each of its building blocks from the perspective of the reviewed papers, providing novel categories and systematization. At the same time, we highlight the main gaps identified in the literature for which further research is needed. To name a few, we identified that some tasks have been widely investigated while others remain unexplored, or methodologies and steps that can be applied to several different tasks are each time reinvented from scratch.

Additionally we explore the applicability of recent NLP techniques based on the transformer architecture to a novel task: detection of debug information bugs in optimized binaries. The correctness of debug information included in optimized binaries is an important problem since most of the software running in production derived from an optimizing compiler. All the current solutions to this problem rely on human-defined rules (i.e. invariants) that once triggered may reveal the presence of a bug. A downside of this approach is that new rules are needed to discover new categories of bugs. Thus, in Chapter 4 we investigated the feasibility of using the Transformer model to discover incorrect debug information. Our results show that our model, named Neuro-Debug^2, is capable of discovering bugs in both synthetic and real datasets. Additionally, we reported 12 unknown bugs in a recent version of LLVM toolchain, 2 of which have been confirmed.

Finally, we explore the applicability of deep learning techniques also for the creation of a binary code model capable of solving multiple downstream tasks (Chapter 5). We created a model, named BinBert, built on a transformer pre-trained on a huge dataset of both assembly instruction sequences and symbolic execution information. Through fine-tuning, BinBert learns how to apply the general

knowledge acquired with pre-training to the specific task. We evaluate BinBert on both standard and novel tasks, showing that it is capable of achieving state of the art perfomances.

6.2 Future Works

Starting from the gaps proposed in Chapter 3 and from the research papers published with this thesis, we identified different future research directions that can be further investigated:

- Conduct a comprehensive study comparing the various binary representations identified in this work and study their effectiveness across multiple tasks. In this regard, it is worth further investigating the impact of execution-based information in the binary representation. In fact, since most studies do not include such information in their representations, they offer capabilities similar to those of static analysis, leaving the potential of execution-based information in binary underexplored. While some effort has been made in this thesis with BinBert, further research is needed. It is imperative that such research uses ablation study to identify specifically when execution information improve on more straightforward approaches;
- Investigate the impact of the different tokenization and preprocessing rules. Although some efforts have been made to study the effects of tokenization and preprocessing on assembly code cite [82], also by BinBert, more studies are necessary;
- Explore the full potential of automatically learned features in binary code analysis tasks, investigating their generalizability across different tasks and datasets, and determining whether they can consistently outperform manually crafted features in various real-world scenarios;
- Determine whether custom neural architectures are necessary for specific downstream tasks or if there exists a specific standard architecture that can effectively represent binary code and address multiple tasks with good performances;
- Make an evaluation of different pre-training tasks trying to understand which one contribute more to create a richer pre-trained binary representation;
- Investigate the perfomance of Neuro-Debug[2] to different compiler-debugger pairs (e.g. gcc and gdb);
- Investigate the perfomance of BinBert to multiple architectures and different tasks.

Finally, given the recent success of Large Language Models [74, 109, 111] in the NLP community, we believe that it is pivotal to understand whether such models are capable of solving binary code analysis better than smaller models. In this

regards, [118] have recently shown that the OpenaAI Codex model [1] is capable of solving some reverse engineering tasks in a zero-shot setting. Further research is needed to highlight whether more modern LLMs are capable of solving multiple reverse engineering tasks or whether smaller or binary specific models are needed.

[1] https://openai.com/index/openai-codex/

Bibliography

[1] Copilot. https://github.com/features/copilot.

[2] Dblp: computer science bibliography. https://dblp.org/.

[3] Google scholar. https://scholar.google.com/.

[4] Ahn, S., Ahn, S., Koo, H., and Paek, Y. Practical binary code similarity detection with bert-based transferable similarity learning. In *38th Annual Computer Security Applications Conference (ACSAC '22)*, ACSAC '22, p. 361–374 (2022).

[5] Alipour, M. A., Groce, A., Gopinath, R., and Christi, A. Generating focused random tests using directed swarm testing. In *2016 International Symposium on Software Testing and Analysis*, ISSTA '16, p. 70–81 (2016).

[6] Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, **51** (2018), 1.

[7] Alrabaee, S., Choo, K.-K. R., Qbea'h, M., and Khasawneh, M. Bindeep: Binary to source code matching using deep learning. In *Proceedings of the 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1100–1107 (2021).

[8] Alrabaee, S., Karbab, E. B., Wang, L., and Debbabi, M. Bineye: Towards efficient binary authorship characterization using deep learning. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS '19)*, vol. 11736, pp. 47–67 (2019).

[9] Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., and Bos, H. An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security '16)*, pp. 583–600 (2016).

[10] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS '14)* (2014).

[11] Artuso, F., Di Luna, G. A., Massarelli, L., and Querzoni, L. Function naming in stripped binaries using neural networks. *CoRR*, **abs/1912.07946** (2019).

[12] Artuso, F., Di Luna, G. A., and Querzoni, L. Debugging debug information with neural networks. *IEEE Access*, **10** (2022), 54136.

[13] Artuso, F., Mormando, M., Di Luna, G. A., and Querzoni, L. Binbert: Binary code understanding with a fine-tunable and execution-aware transformer. *IEEE Transactions on Dependable and Secure Computing*, (2024).

[14] Babokin, D., Regehr, J., and Livinskiy, V. Yarpgen: Yet another random program generator. https://github.com/intel/yarpgen (2020). [Online; accessed 27-July-2020].

[15] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations (ICLR '15)* (2015).

[16] Balakrishnan, G. and Reps, T. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC 2004)*, pp. 5–23.

[17] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *CoRR*, **abs/1610.00502** (2016).

[18] Banerjee, P., Pal, K. K., Wang, F., and Baral, C. Variable name recovery in decompiled binary code using constrained masked language modeling. *CoRR*, **abs/2103.12801** (2021). `arXiv:2103.12801`.

[19] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, p. 845–860 (2014).

[20] Barany, G. Finding missed compiler optimizations by differential testing. In *27th International Conference on Compiler Construction*, CC '18, p. 82–92 (2018).

[21] Bauman, E., Lin, Z., and Hamlen, K. W. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *25th Annual Network and Distributed System Security Symposium (NDSS '18)* (2018).

[22] Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *ArXiv*, **abs/2004.05150** (2020).

[23] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. A neural probabilistic language model. *Machine Learning Research*, **3** (2003), 1137.

[24] Berabi, B., He, J., Raychev, V., and Vechev, M. Tfix: Learning to fix coding errors with a text-to-text transformer. In *38th International Conference on Machine Learning*, vol. 139 of *MLR '21*, p. 780–791 (2021).

[25] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, p. 2329–2344 (2017).

[26] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. *CoRR*, **abs/1607.04606** (2016). arXiv:1607.04606.

[27] Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., and Shah, R. Signature verification using a "siamese" time delay neural network (nips'93). In *Proceedings of the 6th Advances in Neural Information Processing Systems (NIPS '93)*, pp. 737–744 (1993).

[28] Camacho-Collados, J. and Navigli, R. Find the word that does not belong: A framework for an intrinsic evaluation of word vector representations. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pp. 43–50 (2016).

[29] Cao, Y., Liang, R., Chen, K., and Hu, P. Boosting neural networks to decompile optimized binaries. In *38th Annual Computer Security Applications Conference*, ACSAC '22, p. 508–518 (2022).

[30] Chen, J. Learning to accelerate compiler testing. In *40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, p. 472–475 (2018).

[31] Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., and Zhang, L. A survey of compiler testing. *ACM Computing Surveys*, **53** (2020), 1 .

[32] Chen, Q., Lacomis, J., Schwartz, E. J., Le Goues, C., Neubig, G., and Vasilescu, B. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security '22)*, pp. 4327–4343 (2022).

[33] Chen, T. Y., Cheung, S. C., and Yiu, S. M. Metamorphic testing: A new approach for generating next test cases. *ArXiv*, **abs/2002.12543** (2020).

[34] Chen, W., Su, Y., Shen, Y., Chen, Z., Yan, X., and Wang, W. Y. How large a vocabulary does text classification need? a variational approach to vocabulary selection. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (ACL '19)*, vol. 1, pp. 3487–3497 (2019).

[35] Chen, Y., Groce, A., Zhang, C., Wong, W. K., Fern, X., Eide, E., and Regehr., J. Taming compiler fuzzers. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, p. 197–208 (2013).

[36] Chen, Y., Shi, Z., Li, H., Zhao, W., Liu, Y., and Qiao, Y. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *Proceedings of the 2018 Intelligent Systems and Applications (IntelliSys '18)* (edited by K. Arai, S. Kapoor, and R. Bhatia), pp. 35–47 (2018).

[37] Chowdhury, S. A., Shrestha, S. L., Johnson, T. T., and Csallner, C. Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink. In *42nd International Conference on Software Engineering*, ICSE '20, p. 335–346 (2020).

[38] Chua, Z. L., Shen, S., Saxena, P., and Liang, Z. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX '17)*, pp. 99—-116 (2017).

[39] Cummins, C., Petoumenos, P., Murray, A., and Leather, H. Compiler fuzzing through deep learning. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '18, p. 95–105 (2018).

[40] Dai, H., Dai, B., and Song, L. Discriminative embeddings of latent variable models for structured data. In *33rd International Conference on International Conference on Machine Learning (ICML'16)*, vol. 48, p. 2702–2711 (2016).

[41] David, Y., Alon, U., and Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, **4** (2020), 1.

[42] David, Y., Partush, N., and Yahav, E. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, p. 266–280 (2016).

[43] Deshpande, C., Gens, D., and Franz, M. Stackbert: Machine learning assisted static stack frame size recovery on stripped and optimized binaries. In *14th ACM Workshop on Artificial Intelligence and Security (AISec '21)*, p. 85–95 (2021).

[44] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, (2018).

[45] Di Luna, G. A., Italiano, D., Massarelli, L., Osterlund, S., Giuffrida, C., and Querzoni, L. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, p. 1034–1045 (2021).

[46] "Ding, S. H. H., Fung, B. C. M., and Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*, pp. 472–489 (2019).

[47] Drain, D., Wu, C., Svyatkovskiy, A., and Sundaresan, N. Generating bug fixes using pretrained transformers. In *5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS '21, p. 1–8 (2021).

[48] D'Silva, V., Payer, M., and Song, D. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops*, SPW '15, pp. 73–87 (2015).

[49] Duan, Y., Li, X., Wang, J., and Yin, H. Deepbindiff: Learning program-wide code representations for binary diffing. In *27th Annual Network and Distributed System Security Symposium (NDSS '20)* (2020).

[50] Durfina, L., Kroustek, J., and Zemek, P. Psybot malware: A step-by-step decompilation case study. In *20th Working Conference on Reverse Engineering, (WCRE '13)*, pp. 449–456 (2013).

[51] DWARF Standards Committee. The dwarf debugging standard. http://dwarfstd.org/ (2020). [Online; accessed August-2021].

[52] Estrela, V. V. and Hemanth, J. *Deep Learning for Image Processing Applications*. IOS Press (2017).

[53] Evans, J. P., Dannehl, M., and Kinder, J. XFL: extreme function labeling. *CoRR*, **abs/2107.13404** (2021). Available from: https://arxiv.org/abs/2107.13404, `arXiv:2107.13404`.

[54] Fan, W., Ma, Y., Li, Q., Wang, J., Cai, G., Tang, J., and Yin, D. A graph neural network framework for social recommendations. *IEEE Transactions on Knowledge and Data Engineering*, **34** (2022), 2033.

[55] Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., and Zhao, J. Coda: An end-to-end neural program decompiler. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS '19)*, vol. 32, pp. 3703–3714 (2019).

[56] Gao, H., Cheng, S., Xue, Y., and Zhang, W. 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 '21)*, ISSTA 2021, p. 607–619 (2021).

[57] Gao, J., Yang, X., Fu, Y., Jiang, Y., and Sun, J. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE '18)*, pp. 896–899 (2018).

[58] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *34th International Conference on Machine Learning (ICML'17)*, vol. 70 of *ICML'17*, p. 1263–1272 (2017).

[59] Gligorijević, V., et al. Structure-based protein function prediction using graph convolutional networks. *Nature*, **12** (2021).

[60] Gray, J., Sgandurra, D., Cavallaro, L., and Blasco Alis, J. Identifying authorship in malicious binaries: Features, challenges & datasets. *ACM Comput. Surv.*, **56** (2024).

[61] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. Swarm testing. In *2012 International Symposium on Software Testing and Analysis*, ISSTA '12, p. 78–88 (2012).

[62] Guo, W., Mu, D., Xing, X., Du, M., and Song, D. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX '19)*, pp. 1787–1804 (2019).

[63] Guo, Z. and Wang, H. A deep graph neural network-based mechanism for social recommendations. *IEEE Transactions on Industrial Informatics*, **17** (2021), 2776.

[64] Gupta, R., Pal, S., Kanad, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. In *31th AAAI Conference on Artificial Intelligence*, AAAI'17, p. 1345–1351 (2017).

[65] Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., and Sikdar, B. A survey on iot security: Application areas, security threats, and solution architectures. *IEEE Access*, **7** (2019), 82721.

[66] He, H., Lin, X., Weng, Z., Zhao, R., Gan, S., Chen, L., Ji, Y., Wang, J., and Xue, Z. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24)* (2024).

[67] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, **abs/1512.03385** (2015). `arXiv:1512.03385`.

[68] Hennessy, J. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **4** (1982), 323.

[69] Hornik, K., Stinchcombe, M. B., and White, H. L. Multilayer feedforward networks are universal approximators. *Neural Networks*, **2** (1989), 359.

[70] Hu, Z., Chen, T., Chang, K.-W., and Sun, Y. Few-shot representation learning for out-of-vocabulary words. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL '19)*, pp. 4102–4112 (2019).

[71] Islam, A. C., Yamaguchi, F., Dauber, E., Harang, R. E., Rieck, K., Greenstadt, R., and Narayanan, A. When coding style survives compilation: De-anonymizing programmers from executable binaries. *CoRR*, **abs/1512.08546** (2015). Available from: http://arxiv.org/abs/1512.08546, `arXiv:1512.08546`.

[72] J. Xu, K. L. and Mao, B. Cross-architecture testing for compiler-introduced security bugs. In *48th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '21 (2021).

[73] Jia, C. and Chan, W. K. Which compiler optimization options should i use for detecting data races in multithreaded programs? In *International Conference on Automation of Software Test*, AST '13, pp. 53–56 (2013).

[74] Jiang, A. Q., et al. Mistral 7b (2023). Available from: https://arxiv.org/abs/2310.06825, `arXiv:2310.06825`.

[75] Jiang, M., Dai, Q., Zhang, W., Chang, R., Zhou, Y., Luo, X., Wang, R., Liu, Y., and Ren, K. A comprehensive study on ARM disassembly tools. *IEEE Trans. Software Eng.*, **49** (2023), 1683.

[76] Jin, X., Pei, K., Won, J. Y., and Lin, Z. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, p. 1631–1645 (2022).

[77] Karampatsis, R. M., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code != big vocabulary: Open-vocabulary models for source code. In *42nd International Conference on Software Engineering*, ICSE '20, p. 1073–1085 (2020).

[78] Katz, D. S., Ruchti, J., and Schulte, E. Using recurrent neural networks for decompilation. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, pp. 346–356 (2018).

[79] KENDALL, M. G. A new measure of rank correlation. *Biometrika*, **30** (1938), 81.

[80] Kim, D., Kim, E., Cha, S. K., Son, S., and Kim, Y. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, **49** (2023), 1661–1682.

[81] Kim, G., Hong, S., Franz, M., and Song, D. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, p. 151–163 (2022).

[82] Kim, H., Bak, J., Cho, K., and Koo, H. A transformer-based function symbol name inference model from an assembly language for binary reversing. In *2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, p. 951–965 (2023).

[83] Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations (ICLR '17)* (2017).

[84] Lacomis, J., Yin, P., Schwartz, E. J., Allamanis, M., Goues, C. L., Neubig, G., and Vasilescu, B. Dire: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, p. 628–639 (2020).

[85] Lample, G. and Conneau, A. Cross-lingual language model pretraining. In *33rd Annual Conference on Neural Information Processing Systems*, NeurIPS '19 (2019).

[86] Le, Q. and Mikolov, T. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML'14)*, vol. 32, p. II–1188–II–1196 (2014).

[87] Le, V., Afshari, M., and Su, Z. Compiler validation via equivalence modulo inputs. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 216–226 (2014).

[88] Le, V., Sun, C., and Su, Z. Finding deep compiler bugs via guided stochastic program mutation. In *2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, p. 386–399 (2015).

[89] Lee, S., Han, H., Cha, S. K., and Son., S. Montage: A neural network language model-guided javascript engine fuzzer. In *29th USENIX Security Symposium*, USENIX '20, p. 2613–2630 (2020).

[90] Lehmann, D. and Pradel, M. Feedback-directed differential testing of interactive debuggers. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC FSE '18, p. 610–620 (2018).

[91] Li, X., Qu, Y., and Yin, H. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, p. 3236–3251 (2021).

[92] Li, Y., Ding, S., Zhang, Q., and Italiano, D. Debug information validation for optimized code. In *41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '20, p. 1052–1065 (2020).

[93] Li, Y., Gu, C., Dullien, T., Vinyals, O., and Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the 36th International Conference on Machine Learning (ICML '19)*, vol. 97, pp. 3835–3845 (2019).

[94] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. S. Gated graph sequence neural networks. In *4th International Conference on Learning Representations (ICLR '16)* (2016).

[95] Liangboonprakong, C. and Sornil, O. Classification of malware families based on n-grams sequential pattern features. In *Proceedings of the 2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA '13)*, pp. 777–782 (2013).

[96] Lidbury, C., Lascu, A., Chong, N., and Donaldson, A. F. Many-core compiler fuzzing. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, p. 65–76 (2015).

[97] Lim, H. and Debray, S. Automated bug localization in jit compilers. In *17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '21, p. page 153–164 (2021).

[98] Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., and Bengio, Y. A structured self-attentive sentence embedding. *CoRR*, **abs/1703.03130** (2017). Available from: https://arxiv.org/abs/1703.03130, arXiv:1703.03130.

[99] Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., and Zou, W. Adiff: Cross-version binary code similarity detection with dnn. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, p. 667–678 (2018).

[100] Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., and Liu, H. A review of android malware detection approaches based on machine learning. *IEEE Access*, **8** (2020), 124579.

[101] Liu, X., Li, X., Prajapati, R., and Wu, D. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Thirty-Third AAAI Conference on Artificial Intelligence*, AAAI '19 (2019).

[102] Liu, Z. and Wang, S. How far we have come: Testing decompilation correctness of c decompilers. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, p. 475–487 (2020).

[103] Luo, Z., Wang, P., Wang, B., Yong, T., Xie, W., Zhou, X., Liu, D., and Lu, K. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *30th Annual Network and Distributed System Security Symposium (NDSS '23)* (2023).

[104] Madsen, A. and Johansen, A. R. Neural arithmetic units. (2020).

[105] Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., and Balzarotti, D. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security '22)*, pp. 2099–2116 (2022).

[106] Massarelli, L., Di Luna, G. A., Petroni, F., Querzoni, L., and Baldoni, R. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2019 Workshop on Binary Analysis Research (BAR '19)* (2019).

[107] Massarelli, L., Di Luna, G. A., Petroni, F., Querzoni, L., and Baldoni, R. Function representations for binary similarity. *IEEE Transactions on Dependable and Secure Computing*, (2021), 1.

[108] Meng, X., P., M. B., and Jun, K. Identifying multiple authors in a binary program. In *Proceedings of the 22nd European Symposium on Research (ESORICS '17)*, vol. 10493, pp. 286–304 (2017).

[109] Meta. The llama 3 herd of models (2024). Available from: https://arxiv.org/abs/2407.21783, arXiv:2407.21783.

[110] Nitin, V., Saieva, A., Ray, B., and Kaiser, G. E. Direct : A transformer-based model for decompiled identifier renaming. In *NLP4PROG* (2021).

[111] OpenAI. Gpt-4 technical report (2024). Available from: https://arxiv.org/abs/2303.08774, arXiv:2303.08774.

[112] Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T., and Ukegawa, H. o-glassesx: Compiler provenance recovery with attention mechanism from a short code fragment. *Proceedings 2020 Workshop on Binary Analysis Research (BAR '20)*, (2020).

[113] Padmanabhuni, B. M. and Tan, H. B. K. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC '15)*, vol. 2, pp. 450–459 (2015).

[114] Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., and Xu, J. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *42nd IEEE Symposium on Security and Privacy (SP '21)*, pp. 833–851 (2021).

[115] Pang, C., Zhang, T., Yu, R., Mao, B., and Xu, J. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX '22)*, pp. 2479–2495 (2022).

[116] Paszke, A., et al. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the Advances in Neural Information Processing Systems (NEURIPS '19)*, pp. 8024–8035 (2019).

[117] Patrick-Evans, J., Cavallaro, L., and Kinder, J. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, ACSAC '20, p. 373–385 (2020).

[118] Pearce, H., Tan, B., Krishnamurthy, P., Khorrami, F., Karri, R., and Dolan-Gavitt, B. Pop quiz! can a large language model help with reverse engineering? (2022). Available from: https://arxiv.org/abs/2202.01142, arXiv:2202.01142.

[119] Pei, K., Guan, J., Williams-King, D., Yang, J., and Jana, S. XDA: accurate, robust disassembly with transfer learning. In *28th Annual Network and Distributed System Security Symposium (NDSS '21)* (2021).

[120] Pei, K., She, D., Wang, M., Geng, S., Xuan, Z., David, Y., Yang, J., Jana, S., and Ray, B. Neudep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*, p. 747–759 (2022).

[121] Pei, K., Xuan, Z., Yang, J., Jana, S., and Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *CoRR*, **abs/2012.08680** (2020). Available from: https://arxiv.org/abs/2012.08680, arXiv:2012.08680.

[122] Pei, K., et al. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), p. 690–702 (2021).

[123] Pennington, J., Socher, R., and Manning, C. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*, pp. 1532–1543 (2014).

[124] Perozzi, B., Rami, A., and Skiena, S. Deepwalk: online learning of social representations. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*, pp. 701–710 (2014).

[125] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '18)*, vol. 1, pp. 2227–2237 (2018).

[126] Pizzolotto, D. and Inoue, K. Identifying compiler and optimization options from binary code using deep learning approaches. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*, pp. 232–242 (2020).

[127] Qasem, A., Debbabi, M., Lebel, B., and Kassouf, M. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS '23)*, p. 443–456 (2023).

[128] Redmond, K., Luo, L., and Zeng, Q. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *Proceedings of the Workshop on Binary Analysis Research 2019 (BAR '19)* (2019).

[129] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. Test-case reduction for c compiler bugs. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pp. 335–346. Association for Computing Machinery (2012).

[130] Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP '19)*, pp. 3980–3990 (2019).

[131] Rosenblum, N. E., Zhu, X., and Miller, B. P. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS '11)*, vol. 6879, pp. 172–189 (2011).

[132] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional

networks. In *The Semantic Web: 15th International Conference (ESWC '18)*, p. 593–607 (2018).

[133] Shalev, N. and Partush, N. Binary similarity detection using machine learning. PLAS '18, p. 42–47 (2018).

[134] Shen, Q., Ma, H., Chen, J., Tian, Y., Cheung, S. C., and Chen., X. A comprehensive study of deep learning compiler bugs. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, p. 968–980 (2021).

[135] Shin, E. C. R., Song, D., and Moazzezi, R. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security '15)*, pp. 611–626 (2015).

[136] Shoshitaishvili, Y., et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P '16)*, pp. 138–157 (2016).

[137] Song, Q., Zhang, Y., Ouyang, L., and Chen, Y. Binmlm: Binary authorship verification with flow-aware mixture-of-shared language model. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '22)*, pp. 1023–1033.

[138] Sun, C., Le, V., and Su, Z. Finding compiler bugs via live code mutation. In *2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, p. 849–863 (2016).

[139] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *27th International Conference on Neural Information Processing Systems*, vol. 2 of *NIPS'14*, p. 3104–3112 (2014).

[140] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *27th International Conference on Neural Information Processing Systems*, vol. 2 of *NIPS'14*, p. 3104–3112 (2014).

[141] Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. In *53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing* (edited by C. Zong and M. Strube), vol. 1, pp. 1556–1566 (2015).

[142] Talman, A., Yli-Jyrä, A., and Tiedemann, J. Sentence embeddings in nli with iterative refinement encoders. *Natural Language Engineering*, **25** (2019), 467–482.

[143] Tolksdorf, S., Lehmann, D., and Pradel, M. Interactive metamorphic testing of debuggers. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, p. 273–283 (2019).

[144] Tomás, M., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations (ICLR '13)* (2013).

[145] Turc, I., Chang, M., Lee, K., and Toutanova, K. Well-read students learn better: On the importance of pre-training compact models (2019). `arXiv:1908.08962`.

[146] Ucci, D., Aniello, L., and Baldoni, R. Survey of machine learning techniques for malware analysis. *Computer Security*, **81** (2019), 123.

[147] van der Maaten, L. and Hinton, G. Visualizing data using t-sne. *Journal of Machine Learning Research*, **9** (2008), 2579.

[148] Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. *ArXiv*, **abs/1904.01720** (2019).

[149] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Neural Information Processing Systems (NISP '17)*, pp. 5998–6008 (2017).

[150] Veličković, P. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, **79** (2023), 102538.

[151] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, (2017).

[152] Vig, J. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL '19)*, pp. 37–42 (2019).

[153] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. 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*, pp. 353–355 (2018).

[154] Wang, H., Ma, P., Wang, S., Tang, Q., Nie, S., and Wu, S. Sem2vec: Semantics-aware assembly tracelet embedding. *ACM Transaciton Software Engineerinh Methodology*, (2022).

[155] Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., and Zhang, C. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, p. 1–13 (2022).

[156] Wang, J., Sharp, M., Wu, C., Zeng, Q., and Luo, L. Can a deep learning model for one architecture be used for others? Retargeted-Architecture binary code analysis. In *32nd USENIX Security Symposium (USENIX '23)*, pp. 7339–7356 (2023).

[157] Wolf, T., et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP '19)*, pp. 38–45 (2020).

[158] Wu, Y., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, **abs/1609.08144** (2016). `arXiv:1609.08144`.

[159] Xu, J., Mu, D., Xing, X., Liu, P., Chen, P., and Mao, B. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX '17)*, pp. 17–32 (2017).

[160] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., and Song, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pp. 363—-376 (2017).

[161] Xue, H., Sun, S., Venkataramani, G., and Lan, T. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, **7** (2019), 65889.

[162] Yang, C., Liu, Z., Zhao, D., Sun, M., and Chang, E. Y. Network representation learning with rich text information. In *24th International Joint Conference on Artificial Intelligence (IJCAI '15)*, pp. 2111–2117 (2015).

[163] Yang, J., Fu, C., Liu, X., Yin, H., and Zhou, P. Codee: A tensor embedding scheme for binary code search. *IEEE Transaction on Software Engineering*, **48** (2022), 2224.

[164] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, p. 283–294 (2011).

[165] Yin, J., Tan, G., Li, H., Bai, X., Wang, Y.-P., and Hu, S.-M. Debugopt: Debugging fully optimized natively compiled programs using multistage instrumentation. *Science of Computer Programming*, **169** (2019), 18 .

[166] Yu, S., Qu, Y., Hu, X., and Yin, H. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX '22)*, pp. 2709–2725 (2022).

[167] Yu, Y., Si, X., Hu, C., and Zhang, J. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, **31** (2019), 1235.

[168] Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., and Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '20)*, vol. 34, pp. 1145–1152 (2020).

[169] Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., and Wu, S. Codecmr: Cross-modal retrieval for function-level binary source code matching. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)* (2020).

[170] Yule, G. U. An introduction to the theory of statistics. *Charles Griffin and company, London*, (1911).

[171] Zaman, K., Sah, M., Direkoglu, C., and Unoki, M. A survey of audio classification using deep learning. *IEEE Access*, **11** (2023), 106620.

[172] Zhao, W. X., et al. A survey of large language models. *CoRR*, **abs/2303.18223** (2023). `arXiv:2303.18223`.

[173] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *AI open*, **1** (2020), 57.

[174] Zhu, W., Feng, Z., Zhang, Z., Chen, J., Ou, Z., Yang, M., and Zhang, C. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 IEEE Symposium on Security and Privacy (SP '23)*, pp. 2357–2374 (2023).

[175] Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., and Zhang, Z. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS '19)* (2019).