UNIVERSITY OF CALIFORNIA
RIVERSIDE

Learning Assembly Language Models for Security Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xuezixiang Li

December 2024

Dissertation Committee:

Dr. Heng Yin, Chairperson
Dr. Zhiyun Qian
Dr. Chinya V. Ravishankar
Dr. Chengyu Song

The Dissertation of Xuezixiang Li is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Heng Yin, for his invaluable guidance, mentorship, and unwavering support throughout my PhD journey. His expertise and encouragement have been instrumental in shaping my research and academic growth. I am also profoundly grateful to my committee members, Dr. Chengyu Song, Dr. Zhiyun Qian, and Dr. Chinya Ravishankar, for their insightful feedback, constructive criticism, and steadfast support. A special thanks goes to my colleagues, Jinghan Wang, Yue Duan, Wei Song, Ju Chen, Jianlei Chi, Jie Hu, Zhenxiao Qi, Zhaoqi Xiao, Sheng Yu, and my friend Mingjun Yin and Guoren Li, for their camaraderie, collaboration, and constant encouragement. Sharing this journey with them has been an incredible experience, and I have learned so much from each of them.

Finally, I would like to thank my parents Youxin Li and Zenghui Xue, for their love, patience, and understanding throughout this process. Their unwavering belief in me has been my greatest source of strength.

This dissertation includes previously published material entitled "PalmTree: Learning an Assembly Language Model for Instruction Embedding" published in the Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. And two materials under submission entitled "On the Effectiveness of Custom Transformers for Binary Analysis", and "ALMOND: Learning an Assembly Language Model for 0-Shot Code Obfuscation Detection"

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Learning Assembly Language Models for Security Applications

by

Xuezixiang Li

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2024
Dr. Heng Yin, Chairperson

Deep learning has proven its effectiveness in a wide range of binary analysis tasks, such as function boundary detection, binary code search, function prototype inference, and value set analysis. Deep learning-based assembly language models, in particular, have garnered significant attention and delivered promising results. This dissertation presents approaches and evaluations for training assembly language models tailored to diverse security applications.

Firstly, to enhance instruction representation and provide additional support for Deep-learning approaches, we introduce PalmTree, a language model designed for generating general-purpose, high-quality instruction embeddings, which can be used to improve the downstream deep-learning models for various binary analysis applications.

Secondly, in light of more transformer-based assembly language models that have been proposed targeting different security downstream applications, each featuring unique architectural modifications and the introduction of novel pre-training tasks, we undertake a comprehensive evaluation of transformer-based models, including PalmTree, and their pre-training tasks in the context of four distinct security applications.

Lastly, based on the insights gained from our evaluations, we outline our forthcoming work, which focuses on a novel zero-shot learning approach for obfuscation detection. This is achieved through the re-use of the pre-training task of the assembly language model, with the expectation that it can deliver comparable performance to other supervised learning approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The widespread integration of software into modern society has revolutionized countless aspects of life, delivering unparalleled convenience and innovation. However, this advancement has also introduced significant security challenges, as increasingly complex software systems provide opportunities for malicious actors to exploit vulnerabilities. In the realm of binary security, assembly language—a low-level code that directly interfaces with hardware—plays a pivotal role in developing secure software and identifying security flaws. Its symbolic representation of machine code instructions makes it indispensable for scenarios where source code is inaccessible, such as analyzing COTS (Commercial Off-The-Shelf) software. Despite its importance, the highly specialized nature of assembly language and the diversity of its syntax across architectures make it one of the most challenging domains for security analysis. Consequently, research and analysis of assembly code have long been a central focus in the field of binary static analysis.

Recent advances in machine learning, particularly in natural language processing (NLP), have inspired novel approaches in cybersecurity by providing powerful tools to automate vulnerability detection and mitigation. Transformer-based language models, such as BERT [49] and GPT [169], have achieved groundbreaking success in natural language understanding and generation. These models leverage self-attention mechanisms to capture contextual relationships in text more effectively than previous architectures like RNNs and CNNs, enabling significant improvements in tasks such as sentiment analysis, machine translation, and question-answering. Their ability to understand and generate human language has been transformative, setting new benchmarks across numerous NLP tasks.

The success of transformer-based models in NLP has sparked interest in applying these techniques to programming and assembly languages for binary analysis, but significant challenges remain. Unlike natural language, which is rich in semantic context and redundancy, assembly language is concise, rigid in syntax, and lacks high-level abstractions such as variable names or comments, making it harder for models to infer context. Its diversity across architectures (e.g., x86, ARM, MIPS) adds complexity, as each has unique instruction sets and conventions. Additionally, the assembly code's non-linear control flow, driven by jumps and calls, contrasts with the linear structure of natural language sentences, complicating sequence modeling. Dataset limitations, including scarcity and the expertise required for annotation, further hinder model training. Despite these challenges, the structured and deterministic nature of assembly code offers opportunities for models to learn precise relationships between instructions and their effects, paving the way for advancements in automated binary analysis.

Despite these challenges, assembly code offers unique advantages over natural languages. Unlike natural language, which is ambiguous and influenced by cultural or stylistic factors, assembly language follows rigid, deterministic rules, providing consistent patterns for models to learn. Its structured nature, including loops, jumps, and call instructions, aligns well with transformers' sequence-processing capabilities, allowing them to capture program control flow and operational dependencies. These features make assembly language analysis both challenging and promising for transformer-based models.

Leveraging these unique advantages, this dissertation explores the application of transformer-based language models to assembly language and investigates their use in addressing binary static analysis problems. Specifically, this dissertation evaluates existing transformer-based solutions on multiple downstream tasks, provides several recommendations for the design and training of assembly language models, and proposes methods for instruction representation learning through assembly language models, as well as their integration into end-to-end systems. Furthermore, for the specific security task of obfuscation detection, this work overcomes the limitations of existing supervised learning and fine-tuning approaches by utilizing assembly language models and zero-shot learning, achieving state-of-the-art performance. By advancing these capabilities, this work contributes to the foundation of cybersecurity technology, supporting automatic vulnerability detection, malicious code identification, and reverse engineering processes across various assembly language contexts.

1.1 Thesis Statement

This work aims to advance assembly language models across multiple downstream security tasks. Under this topic, we designed a novel language model tailored for assembly code, developed innovative zero-shot techniques to overcome the limitations of the traditional pre-training fine-tuning paradigm, we also conducted comprehensive surveys and evaluations for the existing assembly language model, and offered our insights and recommendations.

Instruction Representation Learning. We proposed an assembly language model called PalmTree for generating general-purpose instruction embeddings by conducting self-supervised training on large-scale unlabeled binary corpora. PalmTree utilizes three pre-training tasks to capture various characteristics of assembly language. These training tasks overcome the problems in existing schemes, thus can help to generate high-quality representations. We conduct both intrinsic and extrinsic evaluations and compare PalmTree with other instruction embedding schemes. PalmTree has the best performance for intrinsic metrics, and outperforms the other instruction embedding schemes for all downstream tasks.

Evaluation of Transformer-Based Models. We evaluate four custom Transformer-based models (i.e. jTrans [202], PalmTree, StateFormer [159], and Trex [158]) and their pre-training tasks on four downstream applications. According to our evaluation results, we have the following surprising observations: aside from MLM (Masked Language Model), many existing pre-training tasks seem either too noisy or too challenging for the Transformer model to learn effectively; the vanilla BERT model is comparable or superior to these custom

Transformers in all the four downstream applications. Moreover, our evaluation suggests that improvements in fine-tuning are generally more beneficial than introducing new pre-training tasks or making architectural modifications. Consequently, we conclude that recent architectural modifications and additional pre-training tasks for Transformer models may offer limited impact that does not sufficiently justify their associated costs.

Zero-Shot Learning for Security Applications. we present ALMOND, a novel zero-shot approach for detecting code obfuscation in binary executables. Unlike previous supervised learning methods, ALMOND does not require labeled obfuscated samples for training. Instead, it leverages a language model pre-trained only on unobfuscated assembly code to identify the linguistic deviations introduced by obfuscation. The key innovation is the use of "error-perplexity" as a detection metric, which focuses on tokens the model fails to predict. Continuous Error Perplexity further enhances this to capture consecutive prediction errors characteristic of obfuscated sequences. Experiments show ALMOND achieves 96.3% accuracy on unseen obfuscation methods, outperforming supervised baselines. On real-world malware samples, it demonstrates an AUC of 0.869 and significantly outperforms the supervise-learning baseline.

Chapter 2

Background

2.1 Instruction Embedding

Instruction embedding, the process of transforming raw assembly instructions into compact, informative numerical representations, has emerged as a pivotal technique for bridging the gap between machine-readable code and human-interpretable semantic understanding. Based on the embedding generation process, existing approaches can be classified into three categories: raw-byte encoding, manually-designed encoding, and learning-based encoding.

Raw-byte Encoding

The most basic approach is to apply a simple encoding on the raw bytes of each instruction, and then feed the encoded instructions into a deep neural network. One such encoding is "one-hot encoding", which converts each byte into a 256-dimensional vector. One of these dimensions is 1 and the others are all 0. MalConv [173] and DeepVSA [77]

take this approach to classify malware and perform coarse-grained value set analysis, respectively. One instruction may be several bytes long. To strengthen the sense of an instruction, DeepVSA further concatenates the one-hot vectors of all the bytes belonging to an instruction, and forms a vector for that instruction.

Shin et al. [185] take a slightly different approach to detect function boundaries. Instead of a one-hot vector, they encode each byte as a 8-dimensional vector, in which each dimension presents a corresponding digit in the binary representation of that byte. For instance, the `0x90` will be encoded as

$$[\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ ]$$

In general, this kind of approach is simple and efficient, because it does not require disassembly, which can be computationally expensive. Its downside, however, is that it does not provide any semantic level information about each instruction. For instance, we do not even know what kind of instruction it is, and what operands it operates on. While the deep neural networks can probably learn some of this information by itself, it seems very difficult for the deep neural networks to completely understand all the instructions.

Manual Encoding of Disassembled Instructions

Knowing that disassembly carries more semantic information about an instruction, this approach first disassembles each instruction and encodes some features from the disassembly.

Li et al. [122] proposed a very simple method, which only extracts opcode to represent an instruction, and encodes each opcode as a one-hot vector. Unfortunately, this

method completely ignores the information from operands. Instruction2Vec [213] makes use of both opcode and operand information. Registers, addresses, and offsets are encoded in different ways, and then concatenated to form a vector representation. Each instruction is encoded as a nine-dimensional feature vector. An instruction is divided into tokens, and tokens are encoded as unique index numbers. While an opcode takes one token, a memory operand takes up to four tokens, including base register, index register, scale, and displacement.

While this approach is able to reveal more information about opcode and operands for each instruction than raw-byte encoding, it does not carry higher-level semantic information about each instruction. For instance, it treats each opcode instruction equally unique, without knowing that `add` and `sub` are both arithmetic operations thus they are more similar to each other than `call`, which is a control transfer operation. Although it is possible to manually encode some of the higher-level semantic information about each instruction, it requires tremendous expert knowledge, and it is hard to get it right.

Learning-based Encoding

Inspired by representation learning in other domains such as NLP (e.g., word2vec [144, 143]), we would like to automatically learn a representation for each instruction that carries higher-level semantic information. Then this instruction-level representation can be used for any downstream binary analysis tasks, achieving high analysis accuracy and generality.

Several attempts have been made to leverage word2vec [144] to automatically learn instruction-level representations (or embeddings), for code similarity detection [224, 141] and function type inference [34], respectively. The basic idea of this approach is to treat

each instruction as a word, and each function as a document. By applying a word2vec algorithm (Skip-gram or CBOW [143, 144]) on the disassembly code in this way, we can learn a continuous numeric vector for each instruction.

In order to detect similar functions in binary code, Asm2Vec [52] makes use of the PV-DM model [115] to generate instruction embeddings and an embedding for the function containing these instructions simultaneously. Unlike the above approach that treats each instruction as a word, Asm2Vec treats each instruction as one opcode and up to two operands and learns embeddings for opcodes and operands separately.

2.1.1 Challenges in Learning-based Encoding

While the learning-based encoding approach seems intriguing, there exist several challenges.

Complex and Diverse Instruction Formats

Instructions (especially those in CISC architectures) are often in a variety of formats, with additional complexities. Listing 2.1 gives several examples of instructions in x86.

In x86, an instruction can have between 0 to 3 operands. An operand can be a CPU register, an expression for a memory location, an immediate constant, or a string symbol. A memory operand is calculated by an expression of "`base`+`index`×`scale`+`displacement`". While `base` and `index` are CPU registers, `scale` is a small constant number and `displacement` can be either a constant number or a string symbol. All these fields are optional. As a result, memory expressions vary a lot. Some instructions have implicit operands. Arithmetic

instructions change EFLAGS implicitly, and conditional jump instructions take EFLAGS as an implicit input.

Listing 2.1: Instructions are complex and diverse

```
; memory operand with complex expression
mov [ebp+eax*4-0x2c], edx
; three explicit operands, eflags as implicit operand
imul [edx], ebx, 100
; prefix, two implicit memory operands
rep movsb
; eflags as implicit input
jne 0x403a98
```

A good instruction-level representation must understand these internal details about each instruction. Unfortunately, the existing learning-based encoding schemes do not cope with these complexities very well. Word2vec, adopted by some previous efforts [224, 141, 34], treats an entire instruction as one single word, totally ignoring these internal details about each instruction.

Asm2Vec [52] looks into instructions to a very limited degree. It considers an instruction having one opcode and up to two operands. In other words, each instruction has up to three tokens, one for opcodes, and up to two for operands. A memory operand with an expression will be treated as one token, and thus it does not understand how a memory address is calculated. It does not take into account other complexities, such as prefix, a third operand, implicit operands, EFLAGS, etc.

Listing 2.2: Instructions can be reordered

```
; prepare the third argument for function call
mov rdx, rbx
; prepare the second argument for function call
mov rsi, rbp
; prepare the first argument for function call
mov rdi, rax
; call memcpy() function
call memcpy
; test rbx register (this instruction is reordered)
test rbx, rbx
; store the return value of memcpy() into rcx register
mov rcx, rax
; conditional jump based on EFLAGS from test instruction
je  0x40adf0
```

Noisy Instruction Context

The context is defined as a small number of instructions before and after the target instruction on the control-flow graph. These instructions within the context often have certain relations with the target instruction, and thus can help infer the target instruction's semantics.

While this assumption might hold in general, compiler optimizations tend to break this assumption to maximize instruction level parallelism. In particular, compiler optimiza-

Table 2.1: Summary of Approaches

Name	**Encoding**	**Internal Structure**	**Context**	**Disassembly Required**
DeepVSA [77]	1-hot encoding on raw-bytes	no	no	no
Instruction2Vec [213]	manually designed	yes	no	yes
InnerEye [224]	word2vec	no	control flow	yes
Asm2Vec [52]	PV-DM	partial	control flow	yes
PalmTree (this work)	BERT	yes	control flow & data flow	yes

tions (e.g., "-fschedule-insns", "-fmodulo-sched", "-fdelayed-branch" in GCC) seek to avoid stalls in the instruction execution pipeline by moving the load from a CPU register or a memory location further away from its last store, and inserting irrelevant instructions in between.

Listing 2.2 gives an example. The `test` instruction at Line 10 has no relation with its surrounding `call` and `mov` instructions. The `test` instruction, which will store its results into `EFLAGS`, is moved before the `mov` instruction by the compiler, such that it is further away from the `je` instruction at Line 14, which will use (load) the `EFLAGS` computed by the `test` instruction at Line 10. From this example, we can see that contextual relations on the control flow can be noisy due to compiler optimizations.

Note that instructions also depend on each other via data flow (e.g., lines 8 and 12 in Listing 2.2). Existing approaches only work on control flow and ignore this important information. On the other hand, it is worth noting that most existing PTMs cannot deal with the sequence longer than 512 tokens [168] (PTMs that can process longer sequences, such as Transformer XL [42], will require more GPU memory), as a result, even if we directly train these PTMs on instruction sequences with MLM, it is hard for them capture long range data dependencies which may happen among different basic blocks. Thus a new pre-training task capturing data flow dependency is desirable.

2.1.2 Summary of Existing Approaches

Table 2.1 summarizes and compares the existing approaches, with respect to which encoding scheme or algorithm is used, whether disassembly is required, whether instruction internal structure is considered, and what context is considered for learning. In summary, raw-byte encoding and manually-designed encoding approaches are too rigid and unable to convey higher-level semantic information about instructions, whereas the existing learning-based encoding approaches cannot address challenges in instruction internal structures and noisy control flow.

2.2 Assembly Language Models

The Transformer model [197] revolutionized natural language processing (NLP) and has been widely adopted in various domains. When applied to computer security and binary analysis, researchers often modify the Transformer architecture and introduce addi-

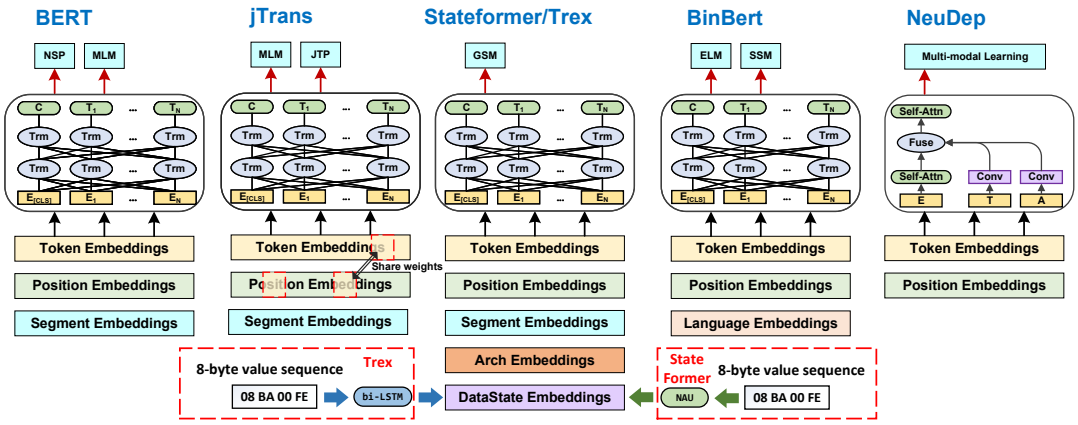


Figure 2.1: Architectural Differences among transformer models

tional pre-training tasks to better capture the unique characteristics of assembly languages. In this section, we survey the use of Transformer models in different research papers, focusing on architectural changes, new pre-training tasks, and downstream tasks.

2.2.1 Architecture

On the one hand, assembly languages are similar to natural languages, because they have their own syntactic and grammatical rules. Consequently, language models like the BERT model depicted in Figure 2.1 can be employed to tackle binary analysis tasks. On the other hand, compared to natural languages, assembly languages are more strictly defined and each instruction has a definite semantic meaning. Furthermore, arithmetic and logic operations, which are rarely found in natural languages, are prevalent in assembly languages. Given these distinctive characteristics of assembly languages, numerous research papers have proposed architectural changes to language models to better capture the syntactic and semantic features. Figure 2.1 depicts the architectures of several prominent Transformer-based models.

StateFormer introduces a numerical representation module that incorporates the Neural Arithmetic Nunit (NAU) [136], which has been proven to be beneficial for capturing the semantics of numerical values involved in arithmetic operations. This module replaces the conventional embedding layer and learns representations for numerical tokens. More specifically, apart from token, position, and segment embeddings, StateFormer introduces the architecture and DataState layers. The architecture layer is to differentiate instruction set architectures (ISAs), as the model is trained for multiple platforms. The DataState layer receives embeddings from the NAU. The embeddings of all five layers are then averaged and fed into the Transformer layers.

Similar to StateFormer, Trex [158] uses the microtrace-based model to generate function token embeddings, but proposed to use a Long Short-Term Memory (LSTM) [97] network to model numerical values involved in arithmetic operations.

jTrans [202] introduces a modification to the positional embedding layer to model the jump instructions and enable ALMs' awareness of control flow information. For each jump pair, its source token's embedding, also called jump embedding, shares parameters with its target token's positional embedding. This design is based on the fact that the source and target of jump instructions are not only as similar as two consecutive tokens, but also have a strong contextual connection. It is worth noting that this architectural modification is exclusive to help the training process. When the trained model is used for inference, this modification is removed.

BinBert [10] concatenates two kinds of inputs: assembly codes and strand-symbolic expressions. A `[SEP]` token is used to distinguish between assembly code and symbolic

expressions. A language embedding layer is also added to the model to differentiate the assembly code and the strand-symbolic expressions. The expressions are generated by a symbolic execution engine which is built on angr [186].

NeuDep [161] further revises the Transformer model. This model acquires the ability to reason about approximate memory dependencies by leveraging the execution behavior of generic binary code during pre-training. To achieve this goal, the authors combined the self-attention layer with the per-byte convolution network by applying a fusion module. The model takes three kinds of sequences as input: the instructions, traces, and code addresses. Instructions are encoded by the self-attention layers, while traces and code addresses are embedded by convolution layers. Subsequently, the fusion module is employed to integrate three embeddings, and the resulting fused embeddings are then passed through an additional self-attention layer for the final encoding. The output of this layer represents the final embedding.

UniASM [74] introduces two pre-training tasks: Assembly Language Generation (ALG) and Similar Function Prediction (SFP). In ALG, two functions compiled from the same source code with different compilation options are treated as a single sentence input, aiming to recover masked tokens based on the first function to teach the model instruction equivalency. SFP uses a batch-wise softmax layer to maximize the similarity between positive pairs and minimize the similarity of negative pairs.

Yu et al. [216] employs four pre-training tasks to capture control flow graph features. In addition to utilizing MLM to capture token-level features, the authors introduced three additional tasks: Adjacency Node Prediction (ANP), Block Inside Graph (BIG), and Graph Classification (GC).

In addition to the papers mentioned above, there exist numerous research papers that combine Transformer-based models with other techniques or models. For instance, BinShot [6] uses DeepSemantic [108] for instruction normalization to alleviate the out-of-vocabulary (OOV) problem. SROBR [193] combines BERT with graph attention networks (GATs) [199] to incorporate control flow features. CodeFormer [126] combines BERT with graph neural networks (GNNs) to capture control flow features. However, they do not introduce any architectural changes or new pre-training tasks to the Transformer model, so they are not the main focus of this paper.

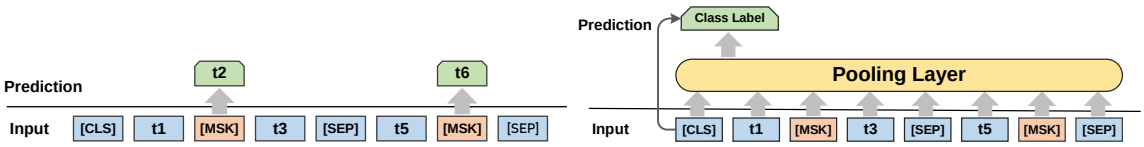
2.2.2 Pre-training Tasks

Pre-training tasks typically involve self-supervised learning on a large corpus which helps a model learn syntactic and semantic information. These tasks can be generally categorized into token-level and sentence-level tasks. Token-level pre-training tasks (depicted in Figure 2.2(a),) usually involve masking certain tokens and requiring the model to predict the masked tokens based on their contextual information. Sentence-level pre-training tasks (depicted in Figure 2.2(b)) employ a pooling mechanism to extract a representation for the entire input. The sentence-level tasks also have a classification head, for training purposes. BERT first introduces the Masked Language Model (MLM), a token-level task, and Next Sentence Prediction (NSP), a sentence-level task, for pre-training. These tasks have demonstrated strong performance in comprehending natural languages. However, assembly languages have some unique characteristics. For instance, certain semantic meanings, such as arithmetic operations and control transfer instructions, cannot be solely learned from the corpus. Some information, such as instruction addresses and lengths, is neither explic-

itly provided nor inferable. These distinctions have prompted various research papers to introduce novel pre-training tasks tailored to assembly languages, with some demonstrating performance enhancements over the standard BERT model for specific tasks.

jTrans [202] introduces Jump Target Prediction (JTP) to help the model learn control flow information. This task requires the model to predict jump targets of randomly selected jump instructions. The nature of this task, which poses a significant challenge even to human experts, requires the model to develop a deep understanding of the control flow, resulting in improved performance.

StateFormer introduces Generative State Modeling (GSM) [159] to teach the model the data and control flow behaviors. This approach involves training the model to predict the changed values of registers and memories after the execution of each instruction. By incorporating this pre-training task, StateFormer ensures that the model understands operational semantics.



(a) Token Level (b) Sequence Level

Figure 2.2: Two types of pre-training tasks

In BinBERT [10], Execution Language Modeling (ELM) is introduced. This pre-training task is similar to MLM. The ELM predicts not only assembly tokens but also tokens in corresponding symbolic expressions. Strand-Symbolic Mapping (SSM) is the other pre-training task proposed by BinBERT. In this task, an instruction strand and a symbolic

expression are provided as inputs, and the model is tasked with determining whether the given symbolic expression belongs to the set of expressions representative of the strand.

PalmTree [119] introduces Context Window Prediction (CWP) and Def-Use Prediction (DUP). CWP predicts whether two given instructions co-occur within a context window to help the model capture implicit control dependencies. DUP focuses on learning the def-use relations between instructions and implicit elements like EFLAGS. This pre-training task is revised from Sentence Ordering Prediction, introduced by Lan et al. [113].

2.2.3 Downtream Tasks

In general, downstream tasks refer to real-world applications or problems that the model is trained to solve, after being pre-trained on a large corpus of data. Downstream tasks are good evaluation methods, enabling us to determine the efficacy of architectural changes and custom pre-training tasks. They also provide insights into the generalizability of these changes and additions.

Function Similarity Search. Function Similarity Search, a.k.a. Binary Code Similarity Detection (BCSD) is one of the most extensively studied downstream tasks in binary analysis and has been evaluated in jTrans [202]. It measures the similarity between a pair of functions and is a building block of various critical research problems such as function name recovery, vulnerability detection, and patch analysis. Similarities can be defined using numerous distance metrics such as cosine distance and Euclidean distance or learned via machine learning models [122]. In this paper, we use cosine distance to measure similarities and consider two functions as similar if they are compiled from the same source code, irrespective of different compilers and compilation options.

In order to learn "similarities", different architectures and objective functions have been proposed. The Siamese network takes function pairs as input and makes positive pairs have the highest similarity while negative pairs have the lowest similarity. The max margin contrastive loss [81] ensures that the distance between a negative function pair exceeds a certain margin. The triplet loss [182] takes an anchor, a positive, and a negative function as input and tries to maximize the distance between the positive and the negative pair. The Normalized Temperature-scaled Cross-Entropy (NT-Xent) loss [187], on the other hand, takes one positive pair and N negative pairs as input and tries to maximize the distance between the positive pair and all negative pairs. In jTrans [202], the term "contrastive learning" refers to the triplet loss. Nonetheless, in this paper, we use the NT-Xent loss for fine-tuning the function similarity search task, as it has demonstrated superior performance compared to other objective functions [32]. A variant of this downstream task is called Algorithm Classification which is first proposed by TBCNN [150].

Type Inference. Type inference [159, 35, 87, 137] is the process of determining the source-level data types, such as integers, structures, and arrays, that are associated with registers or memory regions. This information is valuable for various binary analysis tasks, including reverse engineering and vulnerability detection. On the other hand, type inference is particularly challenging because the information about data types is lost during compilation. Recovering such information requires a deep understanding of instruction semantics, control flow, and other relevant factors. Consequently, type inference can serve as a metric to measure how well the models understand assembly languages.

Function Name Prediction. As its name suggests, the task predicts function names in stripped binaries. Function names often serve as summaries of function behaviors, and thus are very valuable in various security applications such as reverse engineering and code reuse detection. However, similar to type inference, this task presents significant challenges due to the loss of high-level information during compilation. Constructing meaningful function names requires the model to comprehend instruction behaviors. The function name prediction task has been evaluated by Jin et al. [96].

Function Type Recovery. This is a semantic recovery task to predict the number and primitive types of the arguments of a function. EKLAVYA, introduced by Chua et al. [36], is the first neural network model for this task. Similar to type inference and function name prediction, the lack of high-level information poses a significant challenge in function argument recovery, making it a suitable evaluation metric for assessing the capabilities of models.

Value Set Analysis. This is also a semantic recovery task aimed at identifying the memory alias of assembly code. DEEPVSA by Guo et al. [78] is the first machine-learning approach for this task, and it classifies each accessed memory region into one of the following: stack, heap, or global. Unless the memory addresses have been explicitly specified, inferring memory regions from instruction contexts requires a good understanding of instruction semantics and common memory access patterns and is thus challenging and suitable as an evaluation metric for the models.

Call Graph Recovery (Indirect Jump Prediction). Indirect jumps or calls are commonly used in object-oriented programming languages to enable dynamic function execution

during runtime. However, this practice introduces uncertainty in determining the callees until the program is actually executed, thereby hindering the reconstruction of call graphs (CGs) and applications that rely on CG, for example, binary code similarity detection and data flow analysis. Unfortunately, the existing static and dynamic analysis approaches suffer from low precision or recall. Recently, Zhu et al. [222] demonstrated that this problem can be solved by deep neural networks (DNNs), making it a good candidate to evaluate the models' performance.

2.3 Code Obfuscation Detection

Code obfuscation, a common protection technique, transforms code to make it more difficult to understand, analyze, or reverse-engineer without altering its functionality [14, 212]. Obfuscation techniques transform the existing code and introduce redundant or junk code to achieve these goals. Broadly, there are three types of obfuscation techniques: data obfuscation, static code rewriting, and dynamic code rewriting [181].

Data Obfuscation modifies how data is represented or manipulated within the program. Techniques such as encryption or encoding of data—strings, constants, or sensitive variables, make it harder for attackers to reverse-engineer the actual values used during program execution. However, this paper does not cover data obfuscation. Because this type of obfuscation typically does not alter the overall logic of the code. Additionally, this obfuscation does not persist in the binary after compilation, making it undetectable by obfuscation detection tools that analyze binary code.

Static code rewriting is a type of obfuscation that modifies the syntax or control flow structures of code without altering its semantics. Common techniques include control flow obfuscation, string encryption, and code flattening. Strictly speaking, data obfuscation is also a type of static rewriting, but the former primarily focuses on data, while static code rewriting focuses on code. These two categories are also the main focus of this study.

Dynamic Code Rewriting tries to obfuscate the actual execution of the code while still achieving the intended functionality. By doing so, dynamic obfuscation makes it particularly difficult for debuggers or static analysis tools to analyze or trace execution paths. Packers and virtual machine-based obfuscations belong to this category. Dynamic obfuscation techniques are generally difficult for static analysis tools, such as disassemblers, to analyze. This is because the obfuscated code is often only decoded or fully revealed during runtime. Static tools lack the ability to capture the program's runtime behavior, making it hard to reconstruct the original code from a static snapshot. As a result, techniques like dynamic code rewriting are not the primary focus of this paper, as the detection and analysis of these methods require more advanced dynamic analysis approaches rather than static inspection. However, most binaries that use dynamic obfuscation also employ static obfuscation methods to protect their remaining logic. Therefore, static obfuscation detection tools may still be able to detect samples using such techniques.

2.3.1 Obfuscators

An obfuscator is a tool that applies the aforementioned techniques to obfuscate source code or binaries. Typically, an obfuscator offers various obfuscation techniques, allowing developers to use one or combine multiple techniques as needed. Obfuscators can

be categorized into three types based on their target stage [135]: Source Code Obfuscation, Bytecode Obfuscation (Compilation Stage), and Binary Obfuscation. Source code obfuscators, such as Tigress, generate obfuscated source code. These tools can obfuscate not only static languages like C/C++ but also dynamic languages like JavaScript. This paper will focus exclusively on obfuscation detection techniques for C/C++ code. Bytecode obfuscation, also known as compilation-time obfuscation, involves obfuscating intermediate code, such as LLVM IR, during the source code compilation process. For example, OLLVM [99] performs obfuscation at this stage. On the other hand, binary obfuscation tools, such as Alcatraz[1] apply obfuscation methods directly by rewriting the binaries.

2.3.2 Existing Obfuscation Detection Techniques

Obfuscation detection methods can generally be divided into three categories: rule-based approaches, machine learning-based approaches, and deep learning-based approaches. We will discuss each of these methods in detail.

Statistical approaches

As an essential step in obfuscation detection, early methods rely on predefined rules or heuristics to identify patterns or anomalies in the code. Common techniques include the analysis of statistical properties such as entropy, control flow graphs, or n-gram models. To distinguish whether a binary has been packed or encrypted, Lyda et. al. [134] attempted to use entropy as a statistical metric for obfuscation detection. The assumption is that obfuscated binaries exhibit higher entropy than unobfuscated ones due to the randomness

[1]https://github.com/weak1337/Alcatraz

introduced by obfuscation techniques. Kanzaki et al. [100] proposed a new metric called Code Artificiality to determine whether the target code has been obfuscated, which is based on an n-gram model. The intuition is that normal code exhibits predictable patterns of n-grams, while obfuscated code disrupts these patterns. Statistical-based methods are straightforward but often limited in their ability to generalize across different obfuscation techniques.

Machine learning based approaches

In subsequent research, researchers attempted to advance the field by using supervised machine learning methods to classify the specific obfuscation methods applied to binaries [191, 24, 178, 195]. The commonality among these methods is that they treat the obfuscation detection task as a pattern recognition problem, employing machine learning techniques such as Naive Bayes (NB), k-Nearest Neighbor (KNN), Decision Tree (DT), and Random Forest (RF) for supervised training, while introducing innovations in the preprocessing step, specifically in how features were extracted. For instance, Salem et. al. [178] treated disassembly code as text and attempted to use Term Frequency-Inverse Document Frequency (TF-IDF) to extract features. This process generated a feature vector for each program, consisting of the TF-IDF values of the top 128 terms encountered across all disassembly files. This 128-dimensional feature vector is then used as input for training and inference with Naive Bayes (NB) and Decision Tree (DT) models. Tofighi-Shirazi et al. [195] chose to apply static symbolic execution to retrieve the semantic representation of the disassembly code. For feature extraction, they used the Bag of Words (BoW) [138] approach to extract features from the semantic-based raw data for machine learning models. Greco

et al. [72] employed 19 handcrafted features to train machine learning models. By studying the performance of these different features across various obfuscation methods, they aimed to gain a comprehensive understanding of how obfuscation methods affect the properties of target binaries. Last but not least, on the Android platform, AndrODet [149] uses machine learning models to detect three common types of obfuscation techniques in Android applications: identifier renaming, string encryption, and control flow obfuscation.

Deep learning based approaches

With the increasing application of neural networks and deep learning in the security domain, several works have emerged that train neural networks to classify obfuscation methods. For instance, Bacci et al. [12] proposed an LSTM-RNN-based approach to detect seven different Android obfuscation techniques. Zhao et al. [220] designed a composite neural network model. They used a CNN to capture local characteristics and an LSTM to identify the instruction sequence, thereby fully capturing the contextual semantic information of the entire target program. Tian et al. [194] took the research a step further by proposing the Reduced Shortest Path Extraction algorithm, which better samples instruction sequences as input for the neural network. They used a network called BiGRU-CNN for classification, where a GRU is employed to extract features from each reduced shortest path, and a CNN is used for aggregation. Compared to traditional machine learning methods, deep learning approaches mostly utilize learned embeddings rather than manually designed feature vectors. These embeddings not only enhance flexibility and learning efficiency but also improve overall performance.

2.3.3 Challenges

Although the application of deep learning has not only improved the accuracy of obfuscation detection, but learning-based embeddings have also brought flexibility and learning efficiency, the fundamental issue of supervised learning models—generalizability—remains unsolved. Due to the limitations of training data and labels, we can only base our samples and annotations on the data collected. However, in the context of obfuscation detection, the presence of non-public obfuscation tools means that obfuscation detectors must deal with numerous samples obfuscated by unknown methods. Additionally, there is a significant disparity in both the difficulty of obtaining and the number of unobfuscated samples compared to obfuscated samples, leading to dataset imbalance. This poses a major challenge for the training of supervised models.

Chapter 3

Learning an Assembly Language Model for Instruction Embedding

3.1 Introduction

In this chapter, we introduce our first work, named PalmTree, which is a pre-trained assembly language model designed to generate general-purpose instruction embeddings through self-supervised training on large-scale unlabeled binary corpora.

Recently, we have witnessed a surge of research efforts that leverage deep learning to tackle various binary analysis tasks, including function boundary identification [185], binary code similarity detection [207, 224, 124, 216, 158], function prototype inference [34], value set analysis [77], malware classification [173], etc. Deep learning has shown noticeably better performances over the traditional program analysis and machine learning methods.

When applying deep learning to these binary analysis tasks, the first design choice that should be made is: what kind of input should be fed into the neural network model? Generally speaking, there are three choices: we can either directly feed raw bytes into a neural network (e.g., the work by Shin et al. [185], αDiff [124], DeepVSA [77], and MalConv [173]), or feed manually-designed features (e.g., Gemini [207] and Instruction2Vec [213]), or automatically learn to generate a vector representation for each instruction using some representation learning models such as word2vec (e.g., InnerEye [224] and EKLAVYA [34]), and then feed the representations (embeddings) into the downstream models.

Compared to the first two choices, automatically learning instruction-level representation is more attractive for two reasons: (1) it avoids manually designing efforts, which require expert knowledge and may be tedious and error-prone; and (2) it can learn higher-level features rather than pure syntactic features and thus provide better support for downstream tasks. To learn instruction-level representations, researchers adopt algorithms (e.g., word2vec [144] and PV-DM [115]) from Natural Language Processing (NLP) domain, by treating binary assembly code as natural language documents.

Although recent progress in instruction representation learning (instruction embedding) is encouraging, there are still some unsolved problems which may greatly influence the quality of instruction embeddings and limit the quality of downstream models: First, existing approaches ignore the complex internal formats of instructions. For instance, in x86 assembly code, the number of operands can vary from zero to three; an operand could be a CPU register, an expression for a memory location, an immediate constant, or a string symbol; some instructions even have implicit operands, etc. Existing approaches either ignore

this structural information by treating an entire instruction as a word (e.g., InnerEye [224] and EKLAVYA [34]) or only consider a simple instruction format (e.g., Asm2Vec [52]). Second, existing approaches use Control Flow Graph (CFG) to capture contextual information between instructions (e.g., Asm2Vec [52], InnerEye [224], and the work by Yu et al. [216]). However, the contextual information on control flow can be noisy due to compiler optimizations, and cannot reflect the actual dependency relations between instructions.

Moreover, in recent years, pre-trained deep learning models [168] are increasingly attracting attentions in different fields such as Computer Vision (CV) and Natural Language Processing (NLP). The intuition of pre-training is that with the development of deep learning, the numbers of model parameters are increasing rapidly. A much larger dataset is needed to fully train model parameters and to prevent overfitting. Thus, pre-trained models (PTMs) using *large-scale unlabeled corpora* and *self-supervised training* tasks have become very popular in some fields such as NLP. Representative deep pre-trained language models in NLP include BERT [49], GPT [171], RoBERTa [128], ALBERT [114], etc. Considering the naturalness of programming languages [88, 7] including assembly language, it has great potential to pre-train an assembly language model for different binary analysis tasks.

To solve the existing problems in instruction representation learning and capture the underlying characteristics of instructions, in this paper, we propose a pre-trained assembly language model called PalmTree[1] for general-purpose instruction representation learning. PalmTree is based on the BERT [50] model but pre-trained with newly designed training tasks exploiting the inherent characteristics of assembly language.

[1] PalmTree stands for **P**re-trained **A**ssembly **L**anguage **M**odel for Ins**TR**uction **E**mb**E**dding

We are not the first to utilize the BERT model in binary analysis. For instance, Yu et al. [216] proposed to take CFG as input and use BERT to pre-train the token embeddings and block embeddings for the purpose of binary code similarity detection. Trex [158] uses one of BERT's pre-training tasks – Masked Language Model (MLM) to learn program execution semantics from functions' micro-traces (a form of under-constrained dynamic traces) for binary code similarity detection.

Contrast to the existing approaches, our goal is to develop a pre-trained assembly language model for *general-purpose* instruction representation learning. Instead of only using MLM on control flow, PalmTree uses three training tasks to exploit special characteristics of assembly language such as instruction reordering introduced by compiler optimizations and long range data dependencies. The three training tasks work at different granularity levels to effectively train PalmTree to capture internal formats, contextual control flow dependency, and data flow dependency of instructions.

We design a set of intrinsic and extrinsic evaluations to systematically evaluate PalmTree and other instruction embedding models. In intrinsic evaluations, we conduct outlier detection and basic block similarity search. In extrinsic evaluations, we use several downstream binary analysis tasks, which are binary code similarity detection, function type signatures analysis, and value set analysis, to evaluate PalmTree and the baseline models. Experimental results show that PalmTree has the best performance in intrinsic evaluations compared with the existing models. In extrinsic evaluations, PalmTree outperforms the other instruction embedding models and also significantly improves the quality of the downstream applications. We conclude that PalmTree can effectively gen-

erate high-quality instruction embedding which is helpful for different downstream binary analysis tasks.

Experimental results show that PalmTree can provide high-quality general-purpose instruction embeddings. Downstream applications can directly use the generated embeddings in their models. A static embedding lookup table can be generated in advance for common instructions. Such a pre-trained, general-purpose language model scheme is especially useful when computing resources are limited such as on lower-end or embedded devices.

In summary, we have made the following contributions:

- We lay out several challenges in the existing schemes in instruction representation learning.
- We pre-train an assembly language model called PalmTree to generate general-purpose instruction embeddings and overcome the existing challenges.
- We propose to use three pre-training tasks for PalmTree embodying the characteristics of assembly language such as reordering and long range data dependency.
- We conduct extensive empirical evaluations and demonstrate that PalmTree outperforms the other instruction embedding models and also significantly improves the accuracy of downstream binary analysis tasks.
- We plan to release the source code of PalmTree, the pre-trained model, and the evaluation framework to facilitate the follow-up research in this area.

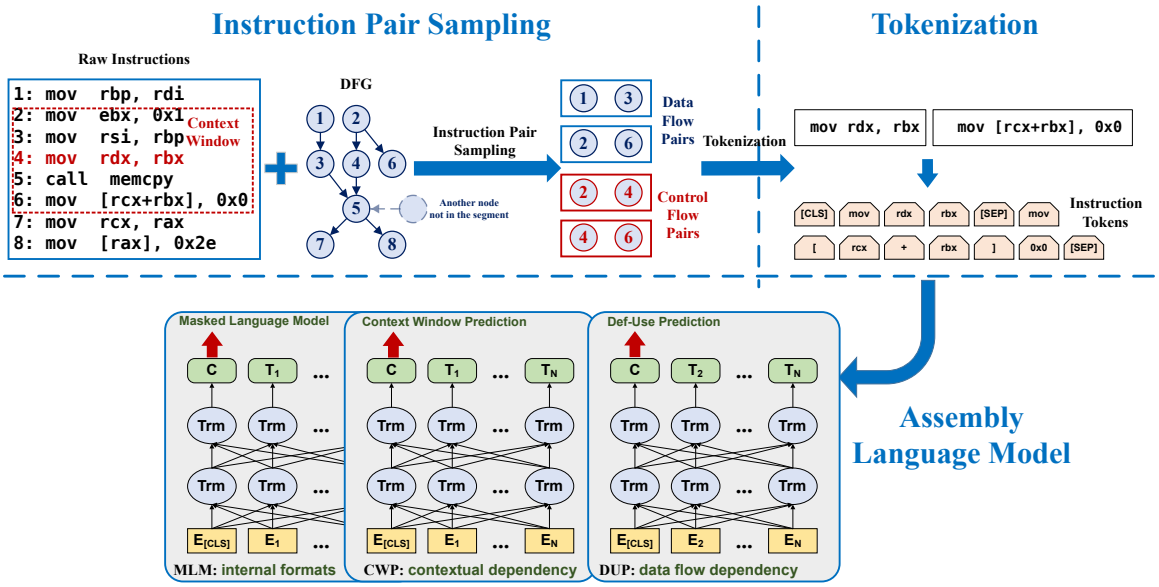


Figure 3.1: System design of PalmTree.

To facilitate further research, we have made the source code and pre-trained PalmTree model publicly available at `https://github.com/palmtreemodel/PalmTree`.

3.2 Design of PalmTree

3.2.1 Overview

To meet the challenges summarized in Section 2.1, we propose PalmTree, a novel instruction embedding scheme that automatically learns a language model for assembly code. PalmTree is based on BERT [50], and incorporates the following important design considerations.

First of all, to capture the complex internal formats of instructions, we use a fine-grained strategy to decompose instructions: we consider each instruction as a sentence and decompose it into basic tokens.

Then, in order to train the deep neural network to understand the internal structures of instructions, we make use of a recently proposed training task in NLP to train the model: Masked Language Model (MLM) [50]. This task trains a language model to predict the masked (missing) tokens within instructions.

Moreover, we would like to train this language model to capture the relationships between instructions. To do so, we design a training task, inspired by word2vec [144] and Asm2Vec [52], which attempts to infer the word/instruction semantics by predicting two instructions' co-occurrence within a sliding window in control flow. We call this training task Context Window Prediction (CWP), which is based on Next Sentence Prediction (NSP) [50] in BERT. Essentially, if two instructions i and j fall within a sliding window in control flow and i appears before j, we say i and j have a contextual relation. Note that this relation is more relaxed than NSP, where two sentences have to be next to each other. We make this design decision based on our observation described in Section 2.1.1: instructions may be reordered by compiler optimizations, so adjacent instructions might not be semantically related.

Furthermore, unlike natural language, instruction semantics are clearly documented. For instance, the source and destination operands for each instruction are clearly stated. Therefore, the data dependency (or def-use relation) between instructions is clearly specified and will not be tampered by compiler optimizations. Based on these facts, we design another training task called Def-Use Prediction (DUP) to further improve our assembly language model. Essentially, we train this language model to predict if two instructions have a def-use relation.

Figure 3.1 presents the design of PalmTree. It consists of three components: Instruction Pair Sampling, Tokenization, and Language Model Training. The main component (Assembly Language Model) of the system is based on the BERT model [50]. `Trm` is the transformer encoder unit, `C` is the hidden state of the first token of the sequence (classification token), T_n $(n = 1 \dots N)$ are hidden states of other tokens of the sequence. After the training process, we use mean pooling of the hidden states of the second last layer of the BERT model as instruction embedding. The Instruction Pair Sampling component is responsible for sampling instruction pairs from binaries based on control flow and def-use relations.

Then, in the second component, the instruction pair is split into tokens. Tokens can be opcode, registers, intermediate numbers, strings, symbols, etc. Special tokens such as strings and memory offsets are encoded and compressed in this step. After that, as introduced earlier, we train the BERT model using the following three tasks: MLM (Masked Language Model), CWP (Context Window Prediction), and Def-Use Prediction (DUP). After the model has been trained, we use the trained language model for instruction em-

bedding generation. In general, the tokenization strategy and MLM will help us address the first challenge in Section 2.1.1, and CWP and DUP can help us address the second challenge.

In Section 3.2.2, we introduce how we construct two kinds of instruction pairs. In Section 3.2.3, we introduce our tokenization process. Then, we introduce how we design different training tasks to pre-train a comprehensive assembly language model for instruction embedding in Section 3.2.4.

3.2.2 Input Generation

We generate two kinds of inputs for PalmTree. First, we disassemble binaries and extract def-use relations. We use Binary Ninja[2] in our implementation, but other disassemblers should work too. With the help of Binary Ninja, we consider dependencies among registers, memory locations, and function call arguments, as well as implicit dependencies introduced by `EFLAGS`. For each instruction, we retrieve data dependencies of each operand, and identify def-use relations between the instruction and its dependent instructions. Second, we sample instruction pairs from control flow sequences, and also sample instruction pairs based on def-use relations. Instruction pairs from control flow are needed by CWP, while instruction pairs from def-use relations are needed by DUP. MLM can take both kinds of instruction pairs.

[2]https://binary.ninja/

3.2.3 Tokenization

As introduced earlier, unlike Asm2Vec [52] which splits an instruction into opcode and up to two operands, we apply a more fine-grained strategy. For instance, given an instruction "`mov rax, qword [rsp+0x58]`", we divide it into "`mov`", "`rax`", "`qword`", "`[`", "`rsp`", "`+`", "`0x58`", and "`]`". In other words, we consider each instruction as a sentence and decompose the operands into more basic elements.

We use the following normalization strategy to alleviate the Out-Of-Vocabulary problem caused by strings and constant numbers. For strings, we use a special token `[str]` to replace them. For constant numbers, if the constants are large (at least five digits in hexadecimal), the exact value is not that useful, so we normalize it with a special token `[addr]`. If the constants are relatively small (less than four digits in hexadecimal), these constants may carry crucial information about which local variables, function arguments, and data structure fields that are accessed. Therefore we keep them as tokens, and encode them as one-hot vectors.

3.2.4 Assembly Language Model

In this section we introduce how we apply the BERT model to our assembly language model for instruction embedding, and how we pre-train the model and adopt the model to downstream tasks.

PalmTree model

Our model is based on BERT [50], the state-of-the-art PTM in many NLP tasks. The proposed model is a multi-layer bidirectional transformer encoder. Transformer, firstly

introduced in 2017 [197], is a neural network architecture solely based on multi-head self attention mechanism. In PalmTree, transformer units are connected bidirectionally and stacked into multiple layers.

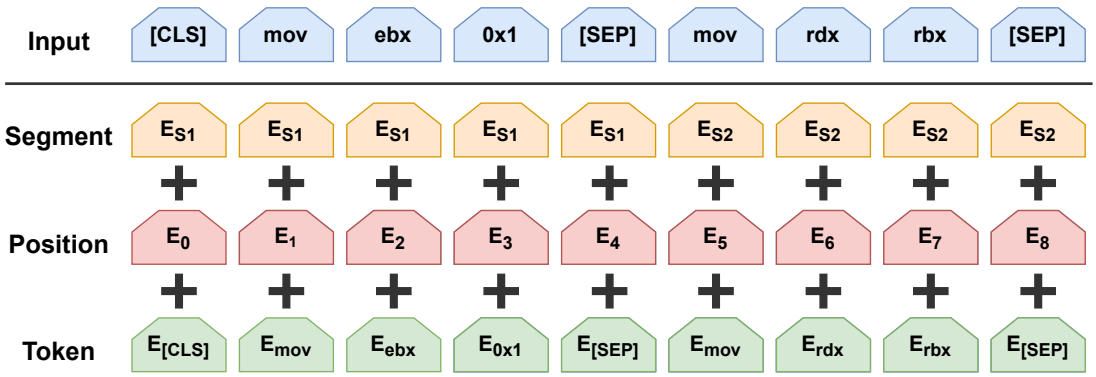


Figure 3.2: Input Representation

We treat each instruction as a sentence and each token as a word. Instructions from control flow and data flow sequences are concatenated and then fed into the BERT model. As shown in Figure 3.2, the first token of this concatenated input is a special token – `[CLS]`, which is used to identify the start of a sequence. Secondly, we use another token `[SEP]` to separate concatenated instructions. Furthermore, we add position embedding and segment embedding to token embedding, and use this mixed vector as the input of the bi-directional transformer network, as shown in Figure 3.2. Position embedding represents different positions in the input sequence, while segment embedding distinguishes the first and second instructions. Position embedding and segment embedding will be trained along with token embeddings. These two embeddings can help dynamically adjust token embeddings according to their locations.

Training task 1: Masked Language Model

The first task we use to pre-train PalmTree is Masked Language Model (MLM), which was firstly introduced in BERT [50]. Here is an example shown in Figure 3.3. Assuming that t_i denotes a token and instruction $I = t_1, t_2, t_3, ..., t_n$ consists of a sequence of tokens. For a given input instruction I, we first randomly select 15% of the tokens to replace. For the chosen tokens, 80% are masked by `[MASK]` (mask-out tokens), 10% are replaced with another token in the vocabulary (corrupted tokens), and 10% of the chosen tokens are unchanged. Then, the transformer encoder learns to predict the masked-out and corrupted tokens, and outputs a probability for predicting a particular token $t_i = [MASK]$ with a softmax layer located on the top of the transformer network:

$$p(\hat{t_i}|I) = \frac{exp(w_i \Theta(I)_i)}{\sum_{k=1}^{K} exp(w_k \Theta(I)_i)} \tag{3.1}$$

where $\hat{t_i}$ denotes the prediction of t_i. $\Theta(I)_i$ is the i^{th} hidden vector of the transformer network Θ in the last layer, when having I as input. and w_i is weight of label i. K is the number of possible labels of token t_i. The model is trained with the Cross Entropy loss function:

$$\mathcal{L}_{MLM} = -\sum_{t_i \in m(I)} \log p(\hat{t}|I) \tag{3.2}$$

where $m(I)$ denotes the set of tokens that are masked.

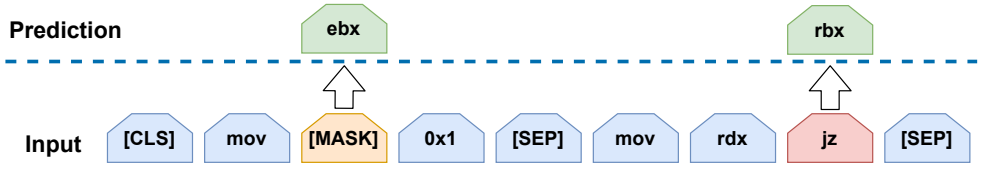


Figure 3.3: Masked Language Model (MLM)

Figure 3.3 shows an example. Given an instruction pair "`mov ebx, 0x1; mov rdx, rbx`", we first add special tokens `[CLS]` and `[SEP]`. Then we randomly select some tokens for replacement. Here we select `ebx` and `rbx`. The token `ebx` is replaced by the `[MASK]` token (the yellow box). The token `rbx` is replaced by the token `jz` (another token in the vocabulary, the red box). Next, we feed this modified instruction pair into the PalmTree model. The model will make a prediction for each token. Here we care about the predictions of the yellow and red boxes, which are the green boxes in Figure 3.3. Only the predictions of those two special tokens are considered in calculating the loss function.

Training task 2: Context Window Prediction

We use this training task to capture control flow information. Many downstream tasks [207, 77, 224, 34] rely on the understanding of contextual relations of code sequences in functions or basic blocks. Instead of predicting the whole following sentence (instruction) [192, 106], we perform a binary classification to predict whether the two given instructions co-occur within a context window or not, which makes it a much easier task compared to the whole sentence prediction. However, unlike natural language, control flows do not have strict dependencies and ordering. As a result, strict Next Sentence Prediction (NSP), firstly proposed by BERT [50], may not be suitable for capturing contextual information of control flow. To tackle this issue, we extend the context window, i.e., we treat each instruction w steps before and w steps after the target instruction in the same basic block as contextually related. w is the context windows size. In Section 3.3.6, we evaluate the performance of different context window sizes, and pick $w = 2$ accordingly. Given an instruction I and a candidate instruction I_{cand} as input, the candidate instruction can be located in

the contextual window of I, or a negative sample randomly selected from the dataset. $\hat{y}$ denotes the prediction of this model. The probability that the candidate instruction I_{cand} is a context instruction of I is defined as

$$p(\hat{y}|I, I_{cand}) = \frac{1}{1 + exp(\Theta(I \parallel I_{cand})_{cls})} \tag{3.3}$$

where $I_{cand} \in \mathbb{C}$, and $\mathbb{C}$ is the candidate set including negative and positive samples. Θ_{cls} is the first output of the transformer network in the last layer. And "$\parallel$" means a concatenation of two instructions. Suppose all instructions belongs to the training set $\mathcal{D}$, then the loss function is:

$$\mathcal{L}_{CWP} = -\sum_{I \in \mathcal{D}} \log p(\hat{y}|I, I_{cand}) \tag{3.4}$$

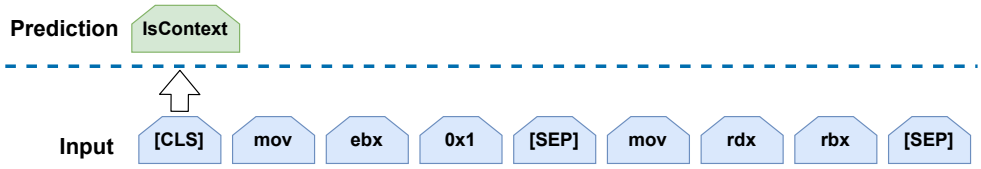


Figure 3.4: Context Window Prediction (CWP)

Here is an example in Figure 3.4. We use the input mentioned above. We feed the unchanged instruction pairs into the PalmTree model and pick the first output vector. We use this vector to predict whether the input are located in the same context window or not. In this case, the two instructions are next to each other. Therefore the correct prediction would be "true".

Training task 3: Def-Use Prediction

To further improve the quality of our instruction embedding, we need not only control flow information but also data dependency information across instructions.

Sentence Ordering Prediction (SOP), first introduced by Lan et al. [114], is a very suitable choice. This task can help the PalmTree model to understand the data relation through DFGs, and we call it Def-Use Prediction (DUP).

Given an instruction pair I_1 and I_2 as input. And we feed $I_1 \parallel I_2$ as a positive sample and $I_2 \parallel I_1$ as a negative sample. $\hat{y}$ denotes the prediction of this model. The probability that the instruction pair is swapped or not is defined as

$$p(\hat{y}|I_1, I_2) = \frac{1}{1 + exp(\Theta(I_1 \parallel I_2)_{cls})} \tag{3.5}$$

where Θ_{cls} is the first output of the transformer network in the last layer. The Cross Entropy loss function is:

$$\mathcal{L}_{DUP} = -\sum_{I \in \mathcal{D}} p(\hat{y}|I_1, I_2) \tag{3.6}$$

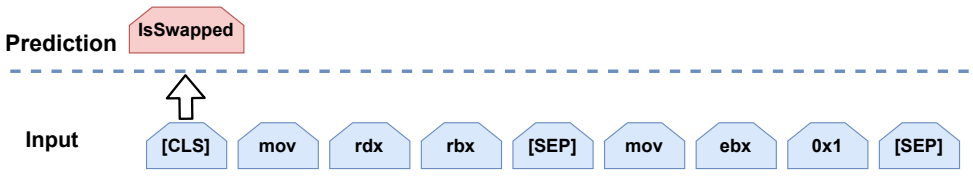


Figure 3.5: Def-Use Prediction (DUP)

We show an example in Figure 3.5. We still use the instruction pair discussed in Figure 3.4, but here we swap the two instructions. So the sequence is "`[CLS] mov`

`rdx rbx [SEP] mov ebx 0x1 [SEP]`". We feed it into PalmTree and use the first output vector to predict whether this instruction pair remains unswapped or not. In this case, it should be predicted as "false" (which means this pair is swapped).

The loss function of PalmTree is the combination of three loss functions:

$$\mathcal{L} = \mathcal{L}_{MLM} + \mathcal{L}_{CWP} + \mathcal{L}_{DUP} \tag{3.7}$$

Instruction Representation

The transformer encoder produces a sequence of hidden states as output. There are multiple ways to generate instruction embeddings from the output. For instance, applying a max/mean pooling. We use mean pooling of the hidden states of the second last layer to represent the whole instruction. This design choice has the following considerations. First, the transformer encoder encodes all the input information into the hidden states. A pooling layer is a good way to utilize the information encoded by transformer. Second, results in BERT [50] also suggest that hidden states of previous layers before the last layer have offer more generalizability than the last layer for some downstream tasks. We evaluated different layer configurations and reported the results in section 3.3.6.

Deployment of the model

There are two ways of deploying PalmTree for downstream applications: *instruction embedding generation*, where the pre-trained parameters are frozen, and *fine-tuning*, where the pre-trained parameters can be further adjusted.

In the first way (instruction embedding generation), PalmTree is used as an off-the-shelf assembly language model to generate high-quality instruction embeddings.

Downstream applications can directly use the generated embeddings in their models. Our evaluation results show that PalmTree without fine-tuning can still outperform existing instruction embedding models such as word2vec and Asm2Vec. This scheme is also very useful when computing resources are limited such as on a lower-end or embedded devices. In this scenario, we can further improve the efficiency by generating a static embedding lookup table in advance. This lookup table contains the embeddings of most common instructions. A trade-off should be made between the model accuracy and the available resources when choosing the lookup table size. A larger lookup table will consume more space but can alleviate the OOV problem (happens when the encountered instruction is not in the table) and improve the accuracy.

In the second way (fine-tuning), PalmTree is fine-tuned and trained together with the downstream model. This scheme will usually provide extra benefits when enough computing resources and training budget are available. There are several fine-tuning strategies [168], e.g., two-stage fine-tuning, multi-task fine-tuning.

3.3 Evaluation

Previous binary analysis studies usually evaluate their approaches by designing specific experiments in an end-to-end manner, since their instruction embeddings are only for individual tasks. In this paper, we focus on evaluating different instruction embedding schemes. To this end, we have designed and implemented an extensive evaluation framework to evaluate PalmTree and the baseline approaches. Evaluations can be classified into two categories: *intrinsic evaluation* and *extrinsic evaluation*. In the remainder of this section,

we first introduce our evaluation framework and experimental configurations, then report and discuss the experimental results.

3.3.1 Evaluation Methodology

Intrinsic Evaluation. In NLP domain, intrinsic evaluation refers to the evaluations that compare the generated embeddings with human assessments [13]. Hence, for each intrinsic metric, manually organized datasets are needed. This kind of dataset could be collected either in laboratory on a limited number of examinees or through crowd-sourcing [129] by using web platforms or offline survey [13]. Unlike the evaluations in NLP domain, programming languages including assembly language (instructions) do not necessarily rely on human assessments. Instead, each opcode and operand in instructions has clear semantic meanings, which can be extracted from instruction reference manuals. Furthermore, debug information generated by different compilers and compiler options can also indicate whether two pieces of code are semantically equivalent. More specifically, we design two intrinsic evaluations: *instruction outlier detection* based on the knowledge of semantic meanings of opcodes and operands from instruction manuals, and *basic block search* by leveraging the debug information associated with source code.

Extrinsic Evaluation. Extrinsic evaluation aims to evaluate the quality of an embedding scheme along with a downstream machine learning model in an end-to-end manner [13]. So if a downstream model is more accurate when integrated with instruction embedding scheme A than the one with scheme B, then A is considered better than B. In this paper, we choose three different binary analysis tasks for extrinsic evaluation, i.e., Gemini [207] for

binary code similarity detection, EKLAVYA [34] for *function type signatures inference*, and DeepVSA [77] for *value set analysis*. We obtained the original implementations of these downstream tasks for this evaluation. All of the downstream applications are implemented based on TensorFlow[3]. Therefore we choose the first way of deploying PalmTree in extrinsic evaluations (see Section 3.2.4). We encoded all the instructions in the corresponding training and testing datasets and then fed the embeddings into downstream applications.

3.3.2 Experimental Setup

Baseline Schemes and PalmTree Configurations. We choose Instruction2Vec, word2vec, and Asm2Vec as baseline schemes. For fair comparison, we set the embedding dimension as 128 for each model. We performed the same normalization method as PalmTree on word2vec and Asm2Vec. We did not set any limitation on the vocabulary size of Asm2Vec and word2vec. We implemented these baseline embedding models and PalmTree using PyTorch [157]. PalmTree is based on BERT but has fewer parameters. While in BERT $\#Layers = 12$, $Head = 12$ and $Hidden\_dimension = 768$, we set $\#Layers = 12$, $Head = 8$, $Hidden\_dimension = 128$ in PalmTree, for the sake of efficiency and training costs. The ratio between the positive and negative pairs in both CWP and DUP is 1:1.

Furthermore, to evaluate the contributions of three training tasks of PalmTree, we set up three configurations:

[3]https://www.tensorflow.org/

- **PalmTree-M**: PalmTree trained with MLM only
- **PalmTree-MC**: PalmTree trained with MLM and CWP
- **PalmTree**: PalmTree trained with MLM, CWP, and DUP

Datasets.

To pre-train PalmTree and evaluate its transferability and generalizability, and evaluate baseline schemes in different downstream applications, we used different binaries from different compilers. The pre-training dataset contains different versions of Binutils[4], Coreutils[5], Diffutils[6], and Findutils[7] on x86-64 platform and compiled with Clang[8] and GCC[9] with different optimization levels. The whole pre-training dataset contains **3,266 binaries** and **2.25 billion** instructions in total. There are about 2.36 billion positive and negative sample pairs during training. To make sure that training and testing datasets do not have much code in common in extrinsic evaluations, we selected completely different testing dataset from different binary families and compiled by different compilers. Please refer to the following sections for more details about dataset settings.

Hardware Configuration. All the experiments were conducted on a dedicated server with a Ryzen 3900X CPU@3.80GHz×12, one GTX 2080Ti GPU, 64 GB memory, and 500 GB SSD.

[4]https://www.gnu.org/software/binutils/
[5]https://www.gnu.org/software/coreutils/
[6]https://www.gnu.org/software/diffutils/
[7]https://www.gnu.org/software/findutils/
[8]https://clang.llvm.org/
[9]https://gcc.gnu.org/

3.3.3 Intrinsic Evaluation

Outlier Detection

In this intrinsic evaluation, we randomly create a set of instructions, one of which is an outlier. That is, this instruction is obviously different from the rest of the instructions in this set. To detect this outlier, we calculate the cosine distance between any two instructions' vector representations (i.e., embeddings), and pick whichever is most distant from the rest. We designed two outlier detection experiments, one for opcode outlier detection, and one for operand, to evaluate whether the instruction embeddings are good enough to distinguish different types of opcodes and operands respectively.

We classify instructions into 12 categories based on their opcode, according to the x86 Assembly Language Reference Manual [154].Table 3.1 shows how we categorize different opcodes by referring to [154]. Table 3.2 shows how we categorize different operand types. The first column shows the type of operands combination. "none" means the instruction has no operand, such as `retn`. "tri" means the instruction has three operands. The other ones are instructions that have two operands. For instance, "reg-reg" means both operands are registers. The type of each operand has been listed in the second and third columns.

We prepared 50,000 instruction sets. Each set consists of four instructions from the same opcode category and one instruction from a different category.

Similarly, we classify instructions based on their operands. Table 3.2 in the Appendix provides details about this process. Essentially, we classify operand lists, according to the number of operands as well as the operand types. We created another 50,000 sets of

Table 3.1: Types of Opcodes

Types	**Opcodes**
Data Movement	mov, push, pop, cwtl, cltq, cqto, cqtd
Unary Operations	inc, dec, neg, not
Binary Operations	lea, leaq, add, sub,imul, xor, or, and
Shift Operations	sal, sar, shr, shl
Special Arithmetic Operations	imulq, mulq, idivq, divq
Comparison and Test Instructions	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
String Instructions	cmps, cmpsb, cmpsl, cmpsw, lods, lodsb, lodsl, lodsw,mov, movsb, movsl, movsw
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 3.2: Types of Operands

Type	**Operand 1**	**Operand 2**	**# of Operands**
none	-	-	0
addr	address	-	1
ref	memory reference	-	1
reg-reg	register	register	2
reg-addr	register	register	2
reg-cnst	register	constant value	2
reg-ref	register	memory reference	2
ref-cnst	memory reference	constant value	2
ref-reg	memory reference	register	2
tri	-	-	3

Table 3.3: Intrinsic Evaluation Results, Stdev. denotes the standard deviation

Model	**opcode outlier**		**operand outlier**		**basicblock sim search**
	Average	Stdev.	Average	Stdev.	AUC
Instruction2Vec	0.863	0.0529	0.860	0.0363	0.871
word2vec	0.269	0.0863	0.256	0.0874	0.842
Asm2Vec	0.865	0.0426	0.542	0.0238	0.894
PalmTree-M	0.855	0.0333	0.785	0.0656	0.910
PalmTree-MC	0.870	0.0449	0.808	0.0435	0.913
PalmTree	**0.871**	0.0440	**0.944**	0.0343	**0.922**

instructions covering 10 categories, and each set contains four instructions coming from the same category, and one from a different category.

The first and second columns of Table 3.3 present the accuracy distributions for opcode outlier detection and operand outlier detection respectively. We can make the following observations: (1) word2vec performs poorly in both experiments, because it does not take into account the instruction internal structures; (2) Instruction2Vec, as a manually-designed embedding, performs generally well in both experiments, because this manual design indeed takes different opcodes and operands into consideration; (3) Asm2Vec performs slightly better than Instruction2Vec in opcode outlier detection, but considerably worse in operand outlier detection, because its modeling for operands is not fine-grained enough; (4) Even though PalmTree-M and PalmTree-MC do not show obvious advantages over Asm2Vec and Instruction2Vec, PalmTree has the best accuracy in both experiments,

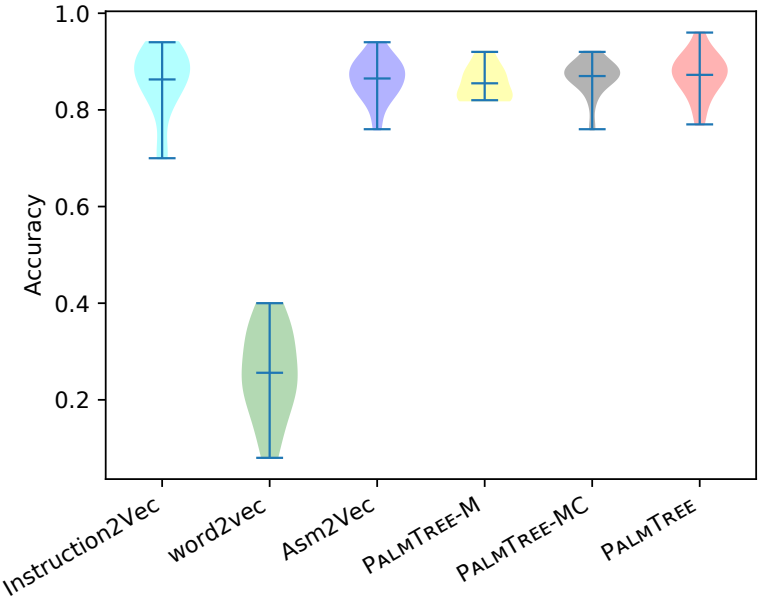


Figure 3.6: Accuracy of Opcode Outlier Detection

which demonstrate that this automatically learned representation can sufficiently capture semantic differences in both opcodes and operands; and (5) All the three pre-training tasks contribute positively to PalmTree in both outlier detection experiments. Particularly, the DUP training task considerably boots the accuracy in both experiments, demonstrating that the def-use relations between instructions indeed help learn the assembly language model. A complete result of outlier detection can be found in Figure 3.6 and Figure 3.7.

Basic Block Search

In this intrinsic evaluation, we compute an embedding for each basic block (a sequence of instructions with only one entry and one exit), by averaging the instruction embeddings in it. Given one basic block, we use its embedding to find semantically equivalent basic blocks based on the cosine distance between two basic block embeddings.

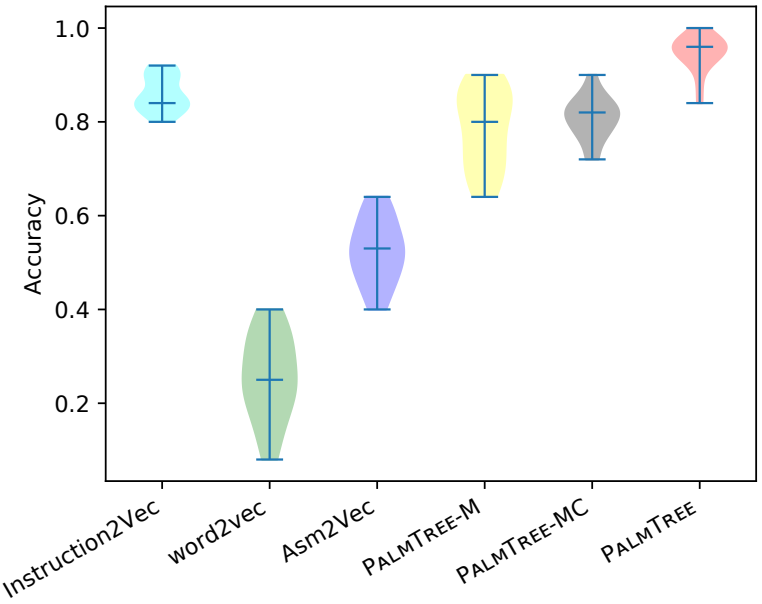


Figure 3.7: Accuracy of Operands Outlier Detection

We use `openssl-1.1.0h` and `glibc-2.29.1` as the testing set, which is not included in our training set. We compile them with O1, O2, and O3 optimization levels. We use the same method used in DeepBinDiff [55], which relies on the debug information from the program source code as the ground truth.

Figure 3.8 shows the ROC curves of Instruction2Vec, word2vec, Asm2Vec, and PalmTree for basic block search. Table 3.3 further lists the AUC (Area Under the Curve) score for each embedding scheme. We can observe that (1) word2vec, once again, has the worst performance; (2) the manually-designed embedding scheme, Instruction2Vec, is even better than word2vec, an automatically learned embedding scheme; (3) Asm2Vec performs reasonably well, but still worse than three configurations of PalmTree; and (4) The three PalmTree configurations have better AUC than other baselines, while consecutive performance improvements are observed.

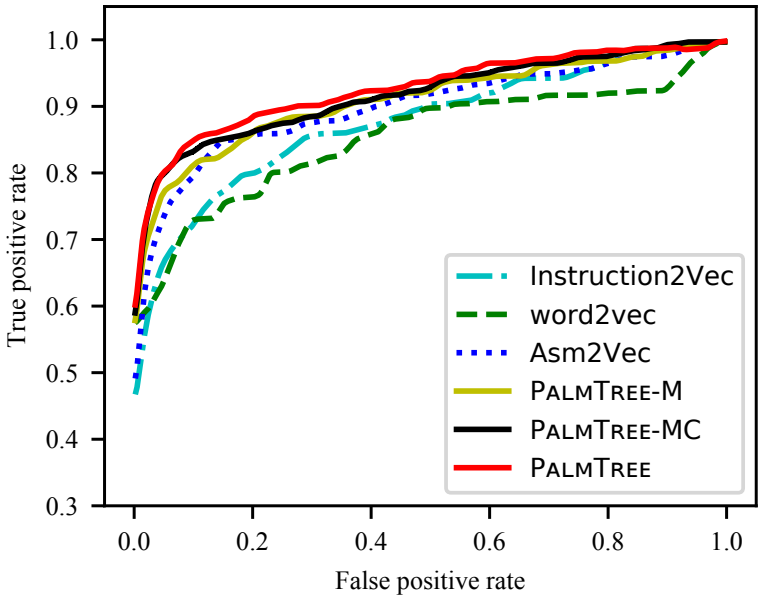


Figure 3.8: ROC curves for Basic Block Search

PalmTree ranks the first in all intrinsic evaluation experiments, demonstrating the strength of the automatically learned assembly language model. And the performance improvements between different PalmTree configurations show positive contributions of individual training tasks.

3.3.4 Extrinsic Evaluation

An extrinsic evaluation reflects the ability of an instruction embedding model to be used as an input of downstream machine learning algorithms for one or several specific tasks [13]. As introduced earlier, we select three downstream tasks in binary analysis field, which are binary code similarity detection, function type signature analysis, and value set analysis.

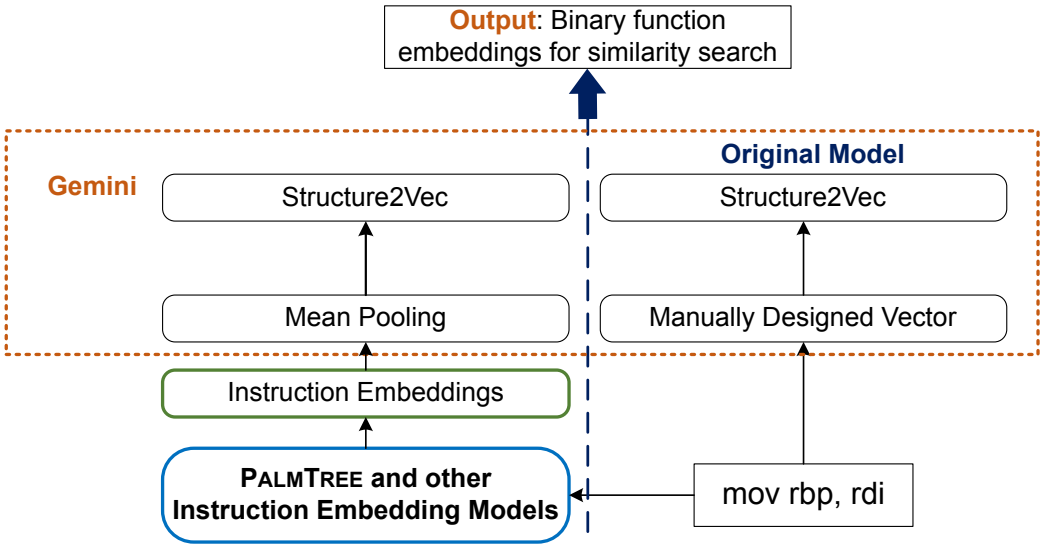


Figure 3.9: Instruction embedding models and the downstream model Gemini

Binary Code Similarity Detection

Gemini [207] is a neural network-based approach for cross-platform binary code similarity detection. The model is based on Structure2Vec [40] and takes ACFG (Attributed Control Flow Graph) as input. In an ACFG, each node is a manually formed feature vector for each basic block. Table 3.4 shows the attributes (i.e., features) of a basic block in the original implementation.

In this experiment, we evaluate the performance of Gemini, when having Instruction2Vec, word2vec, Asm2Vec, PALMTREE-M, PALMTREE-MC, and PALMTREE as input, respectively. Moreover, we also used one-hot vectors with an embedding layer as a kind of instruction embedding (denoted as "one-hot") as another baseline. The embedding layer will be trained along with Gemini. Figure 3.9 shows how we adopt different instruction embedding models to Gemini. Since Gemini takes a feature vector for each basic block, we use

Table 3.4: Attributes of Basic Blocks in Gemini [207]

Type	Attribute name
Block-level attributes	String Constants, Numeric Constants, No. of Transfer Instructions, No. of Calls, No. of Instructions,No. of Arithmetic Instructions
Inter-block attributes	No. of offspring, Betweenness

mean pooling to generate basic block embeddings based on embeddings of the instructions in the corresponding basic block. The architectures of our modified model and the original model are both shown in Figure 3.9. We also included its original basic block features as an additional baseline (denoted as "Gemini") for comparison.

The accuracy of the original Gemini is reported to be very high (with an AUC of 0.971). However, this might be due to overfitting, since the training and testing sets are from OpenSSL compiled by the same compiler Clang. To really evaluate the generalizability (i.e., the ability to adapt to previously unseen data) of the trained models under different inputs, we use `binutils-2.26`, `binutils-2.30`, and `coreutils-8.30` compiled by Clang as training set (237 binaries in total), and used `openssl-1.1.0h`, `openssl-1.0.1`, and `glibc-2.29.1` compiled by GCC as testing set (14 binaries). In other words, the training and testing sets are completely different and the compilers are different too.

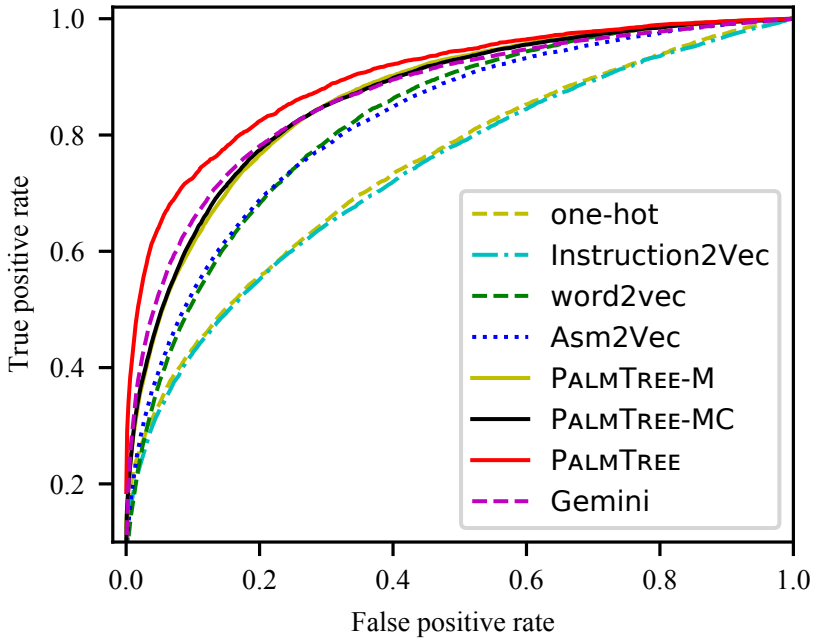


Figure 3.10: ROC curves of Gemini

Table 3.5: AUC values of Gemini

Model	**AUC**	**Model**	**AUC**
one-hot	0.745	Gemini	0.866
Instruction2Vec	0.738	PalmTree-M	0.864
word2vec	0.826	PalmTree-MC	0.866
Asm2Vec	0.823	PalmTree	**0.921**

Table 3.5 gives the AUC values of Gemini when different models are used to generate its input. Figure 3.10 shows the ROC curves of Gemini when different instruction embedding models are used. Based on Table 3.5, we can make the following observations:

(1) Although the original paper [207] reported very encouraging performance of Gemini, we can observe that the original Gemini model does not generalize very well to completely new testing data.

(2) The manually designed embedding schemes, Instruction2Vec and one-hot vector, perform poorly, signifying that manually selected features might be only suitable for specific tasks.

(3) Despite that the testing set is considerably different from the training set, PalmTree can still perform reasonably well and beat the remaining schemes, demonstrating that PalmTree can substantially boost the generalizability of downstream tasks.

(4) All the three pre-training tasks contribute to the final model (PalmTree) for Gemini. However, both PalmTree-M and PalmTree-MC do not show obvious advantages over other baselines, signifying that only the complete PalmTree with the three training tasks can generate better embeddings than previous approaches in this downstream task.

Function Type Signature Inference

Function type signature inference is a task of inferring the number and primitive types of the arguments of a function. To evaluate the quality of instruction embeddings in

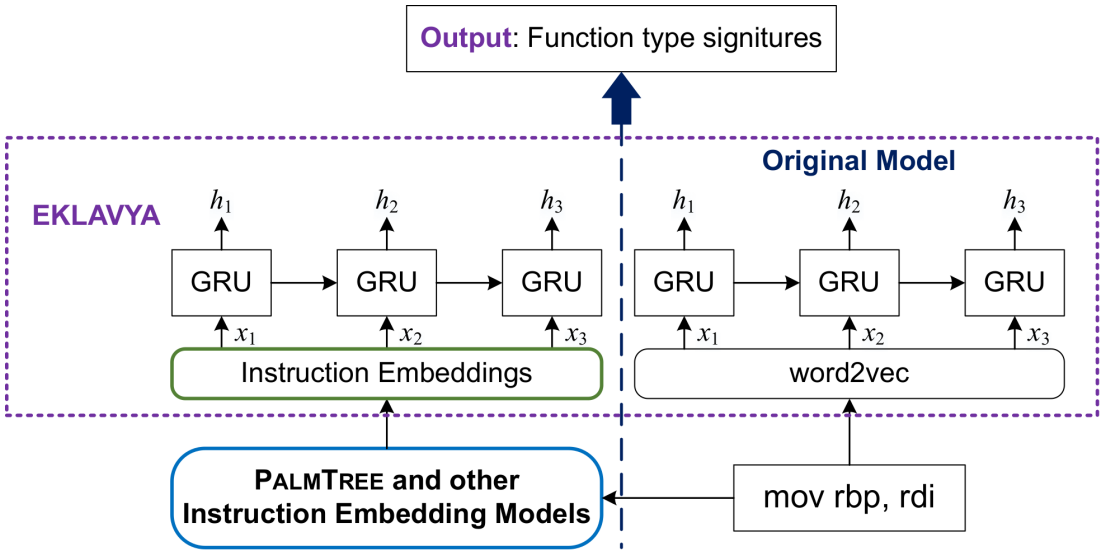


Figure 3.11: Instruction embedding models and EKLAVYA

this task, we select EKLAVYA, an approach proposed by Chua et al. [34]. It is based on a multi-layer GRU (Gated Recurrent Unit) network and uses word2vec as the instruction embedding method. According to the original paper, word2vec was pre-trained with the whole training dataset. Then, they trained a GRU network to infer function type signatures.

In this evaluation, we test the performances of different types of embeddings using EKLAVYA as the downstream application. Since the original model is not an end-to-end model, we do not need an embedding layer between instruction embeddings and the GRU network. We replaced the original word2vec in EKLAVYA with one-hot encoding, Instruction2Vec, Asm2Vec, PALMTREE-M, PALMTREE-MC, and PALMTREE, as shown in Figure 3.11.

Similarly, in order to evaluate the generalizability of the trained downstream models, we used very different training and testing sets (the same datasets described in Section 3.3.4).

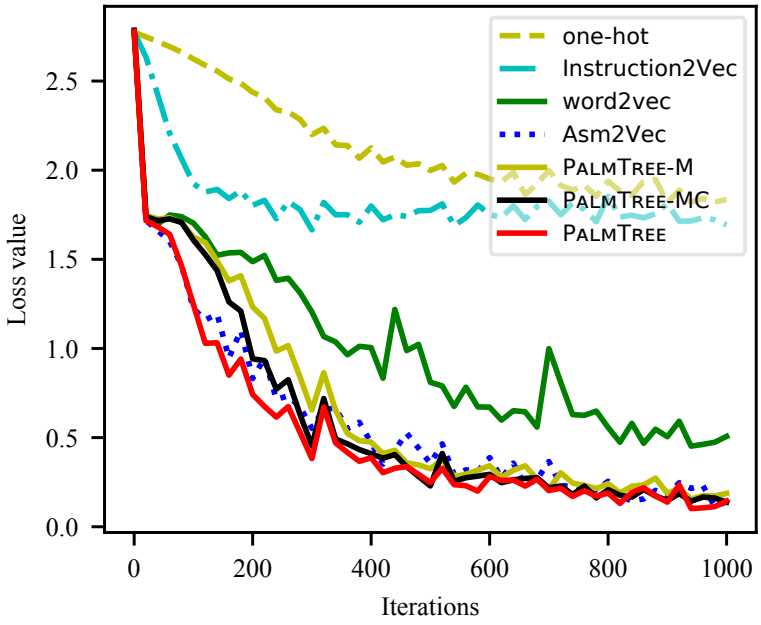


Figure 3.12: Loss value during training

Table 3.6 and Figure 3.14 presents the accuracy of EKLAVYA on the testing dataset. Figure 3.12, and Figure 3.13 shows the loss value and accuracy of EKLAVYA during training and testing. From the results we can make the following observations:

(1) PalmTree and Asm2Vec can achieve higher accuracy than word2vec, which is the original choice of EKLAVYA.

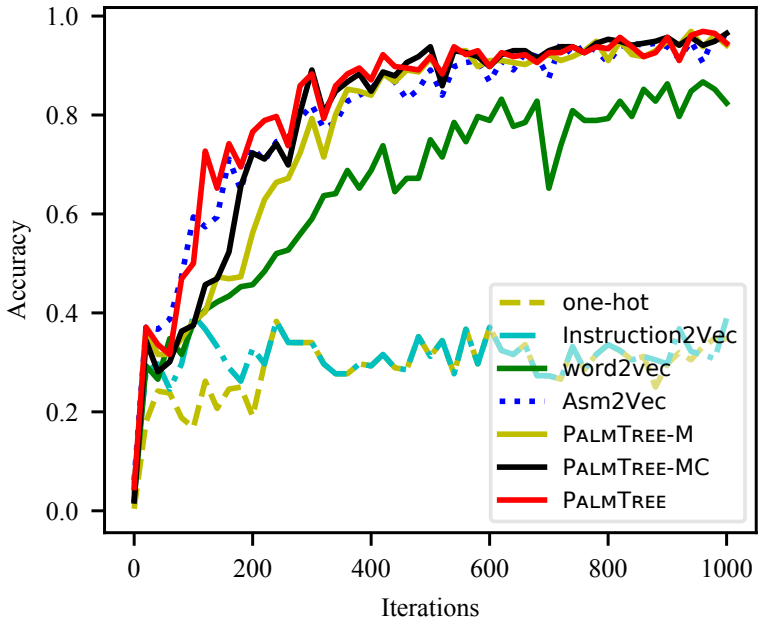


Figure 3.13: Accuracy during training

(2) PalmTree has the best accuracy on the testing dataset, demonstrating that EKLAVYA when fed with PalmTree as instruction embeddings can achieve the best generalizability. Moreover, CWP contributes more (see PalmTree-MC), which implies that control-flow information plays a more significant role in EKLAVYA.

(3) Instruction2Vec performs very poorly in this evaluation, signifying that, when not done correctly, manual feature selection may disturb and mislead a downstream model.

(4) The poor results of one-hot encoding show that a good instruction embedding model is indeed necessary. At least in this task, it is very difficult for the deep neural network to learn instruction semantic through end-to-end training.

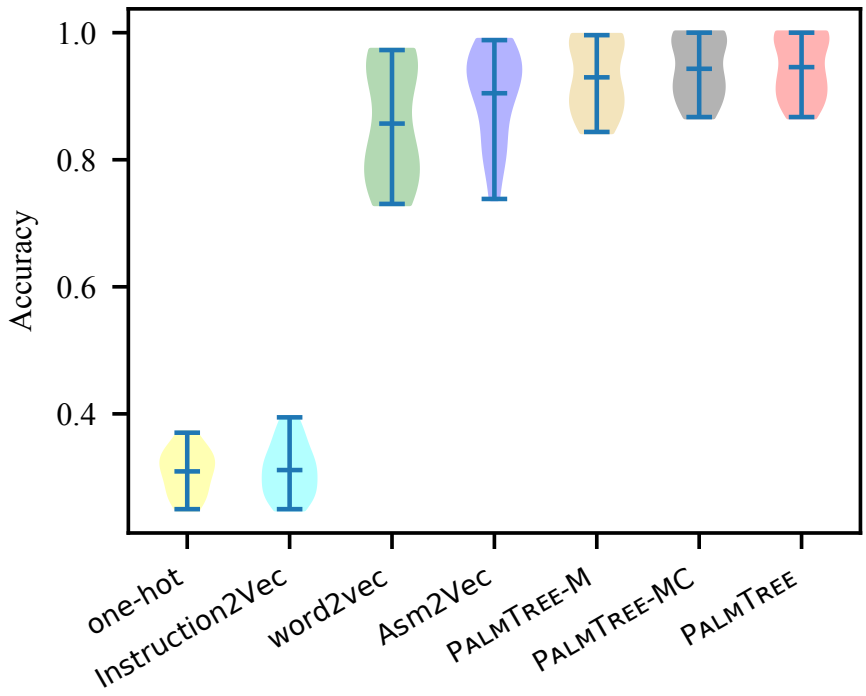


Figure 3.14: Accuracy of EKLAVYA

Value Set Analysis

DeepVSA [77] makes use of a hierarchical LSTM network to conduct a coarse-grained value set analysis, which characterizes memory references into regions like global, heap, stack, and other. It feeds instruction raw bytes as input into a multi-layer LSTM network to generate instruction embeddings. It then feeds the generated instruction representations into another multi-layer bi-directional LSTM network, which is supposed to capture the dependency between instructions and eventually predict the memory access regions.

In our experiment, we use different kinds of instruction embeddings to replace the original instruction embedding generation model in DeepVSA. We use the original training

Table 3.6: Accuracy and Standard Deviation of EKLAVYA

Model	**Accuracy**	**Standard Deviation**
one-hot	0.309	0.0338
Instruction2Vec	0.311	0.0407
word2vec	0.856	0.0884
Asm2Vec	0.904	0.0686
PalmTree-M	0.929	0.0554
PalmTree-MC	0.943	0.0476
PalmTree	**0.946**	0.0475

and testing datasets of DeepVSA and compare prediction accuracy of different kinds of embeddings. The original datasets contain raw bytes only, thus we need to disassemble these raw bytes. After that we tokenize and encode these disassembled instructions for training and testing. We add an embedding layer before the LSTM network to further adjust instruction embeddings, as shown in Figure 3.15.

We use part of the dataset provided by the authors of DeepVSA. The whole dataset provided by the authors has 13.8 million instructions for training and 10.1 million for testing. Our dataset has 9.6 million instructions for training and 4.8 million for testing, due to the disassembly time costs. As explained in their paper [77], their dataset also used Clang and GCC as compilers and had no overlapping instructions between the training and testing datasets.

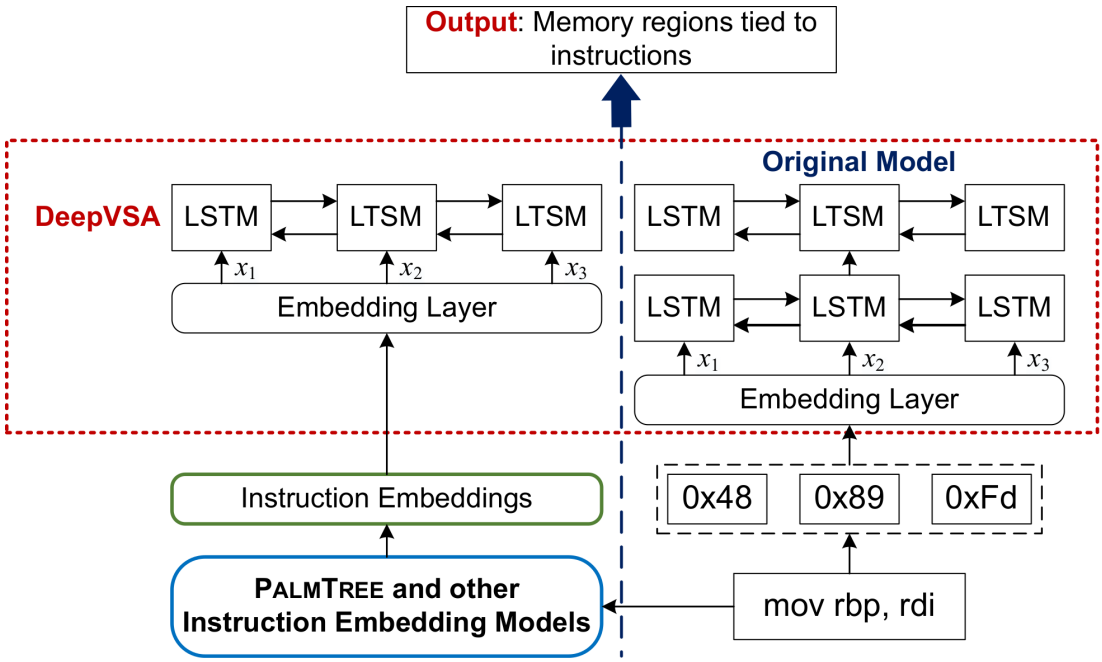


Figure 3.15: Instruction embedding models and the downstream model DeepVSA

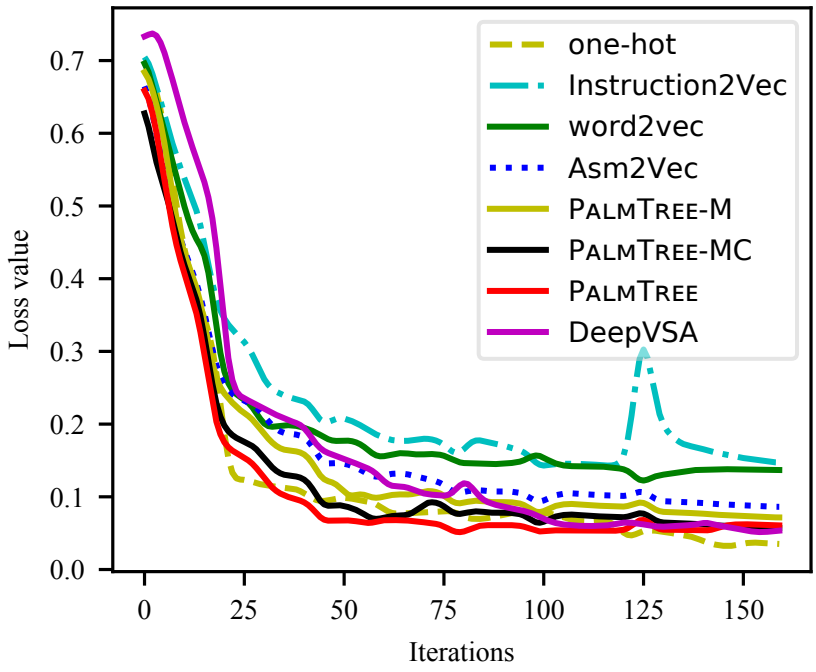


Figure 3.16: Loss value of DeepVSA during training

Table 3.7 lists the experimental results. We use Precision (P), Recall (R), and F1 scores to measure the performance. Figure 3.16 depicts the loss values of DeepVSA during training, when different instruction embedding schemes are used as its input. From these results, we have the following observations:

(1) PalmTree has visibly better results than the original DeepVSA and the other baselines in Global and Heap, and has slightly better results in Stack and Other since other baselines also have scores greater than 0.9.

(2) The three training tasks of PalmTree indeed contribute to the final result. It indicates that PalmTree indeed captures the data flows between instructions. In

Table 3.7: Results of DeepVSA

Embeddings	**Global**			**Heap**			**Stack**			**Other**		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
one-hot	0.453	0.670	0.540	0.507	**0.716**	0.594	0.959	0.866	0.910	0.953	0.965	0.959
Instruction2Vec	0.595	0.726	0.654	0.512	0.633	0.566	0.932	0.898	0.914	0.948	0.946	0.947
word2vec	0.147	0.535	0.230	0.435	0.595	0.503	0.802	0.420	0.776	0.889	0.863	0.876
Asm2Vec	0.482	0.557	0.517	0.410	0.320	0.359	0.928	0.894	0.911	0.933	0.964	0.948
DeepVSA	**0.961**	0.738	0.835	0.589	0.580	0.584	0.974	0.917	0.944	0.943	0.976	0.959
PalmTree-M	0.845	0.732	0.784	0.572	0.625	0.597	0.963	0.909	0.935	0.956	0.969	0.962
PalmTree-MC	0.910	0.755	0.825	**0.758**	0.675	0.714	0.965	0.897	0.929	0.958	**0.988**	**0.972**
PalmTree	0.912	**0.805**	**0.855**	0.755	0.678	**0.714**	**0.974**	**0.929**	**0.950**	**0.959**	0.983	0.971

comparison, the other instruction embedding models are unable to capture data dependency information very well.

(3) PalmTree converged faster than original DeepVSA (see Figure 3.16), indicating that instruction embedding model can accelerate the training phase of downstream tasks.

PalmTree outperforms the other instruction embedding approaches in each extrinsic evaluation. Also, PalmTree can speed up training and further improve downstream models by providing high-quality instruction embeddings. In contrast, word2vec and Instruction2Vec perform poorly in all the three downstream tasks, showing that the poor quality of an instruction embedding will adversely affect the overall performance of downstream applications.

3.3.5 Runtime Efficiency

In this section, we conduct an experiment to evaluate runtime efficiencies of PalmTree and baseline approaches. First, we test the runtime efficiencies of different instruction embedding approaches. Second, we test the runtime efficiency of PalmTree when having different embedding sizes. We use 64, 128, 256, and 512 as embedding sizes, while 128 is the default setting. In the transformer encoder of PalmTree, the width of each feed-forward hidden layer is fixed and related to the size of the final output layer, which is 4 times of the embedding size [114]. We use `Coreutils-8.30` as the dataset. It includes 107 binaries and 1,006,169 instructions. We disassembled the binaries with Binary Ninja and feed them into the baseline models. Due to the limitation of GPU memory, we treated 5,000 instructions as a batch.

Table 3.8: Efficiency of PalmTree and baselines

embedding size	**encoding time**	**throughput (number of instructions/sec)**
Instruction2vec	6.684	150,538
word2vec	0.421	2,386,881
Asm2Vec	17.250	58,328
PalmTree-64	41.682	24,138
PalmTree-128	70.202	14,332
PalmTree-256	135.233	7,440
PalmTree-512	253.355	3,971

Table 3.8 shows the encoding time and throughput of different models when encoding the 107 binaries in `Coreutils-8.30`. From the results, we can make several observations. First, PalmTree is much slower than previous embedding approaches such as word2vec and Asm2Vec. This is expected, since PalmTree has a deep transformer network. However, with the acceleration of the GPU, PalmTree can finish encoding the 107 binaries in about 70 seconds, which is acceptable. Furthermore, as an instruction level embedding approach, PalmTree can have an embedding lookup table as well to store some frequently used embeddings. This lookup table works as fast as word2vec and can further boost the efficiency of PalmTree. Last but not least, from the results we observed that it would be 1.7 to 1.9 times slower when doubling the embedding size.

3.3.6 Hyperparameter Selection

To further study the influences of different hyperparameter configurations of PalmTree, we trained PalmTree with different embedding sizes (64, 128, 256, and 512) and different context window sizes (1, 2, 3, and 4). We also evaluated different output layer configurations when generating instruction embeddings.

Embedding sizes

In this experiment, we evaluate the performance of PalmTree with different embedding sizes. Here we use 64, 128, 256, and 512 as instruction sizes, which is the same as the previous experiment. We test these 4 models on our intrinsic evaluation tasks.

Table 3.9 shows all of the results of intrinsic evaluation when having different embedding sizes. From the results, we can observe that there is a clear trend that the performance becomes better when increasing the embedding size. The largest embedding size has the best performance in all three metrics. However, considering efficiency, we recommend having a suitable embedding size configuration according to the hardware capacities. For example, we only have a single GPU (GTX 2080Ti) in our server, thus we chose 128 as the embedding size.

Output layer configurations

In this experiment, we evaluate the performance of PalmTree with different output layer configurations. It means that we select a different layer of the transformer model as the output of PalmTree. By default, PalmTree uses the second-last layer

Table 3.9: Embedding sizes

Embedding Sizes	**opcode outlier detection**		**operand outlier detecion**		**basicblock sim search**
	Avg.	Stdev.	Avg.	Stdev.	AUC
64	0.836	0.0588	0.940	0.0387	0.917
128	0.871	0.0440	0.944	0.0343	0.922
256	0.848	0.0560	0.954	0.0343	0.929
512	**0.878**	0.0525	**0.957**	0.0335	**0.929**

as the output layer. And we evaluate five different settings, which are the last layer, the second-last layer, the third-last layer, and the fourth-last layer, on our intrinsic evaluation tasks. The embedding size in this experiment is set as 128.

Table 3.10: Output layer configurations

Layers	**opcode outlier detection**		**operand outlier detecion**		**basicblock sim search**
	Avg.	Stdev.	Avg.	Stdev.	AUC
last	0.862	0.0460	**0.982**	0.0140	0.915
2nd-last	**0.871**	0.0440	0.944	0.0343	**0.922**
3rd-last	0.868	0.0391	0.956	0.0287	0.918
4th-last	0.866	0.0395	0.961	0.0248	0.913

Table 3.10 shows all of the results of the intrinsic metrics when having a different layer as the output layer. There is no obvious advantage to choose any layer as the output

layer. However, the second-last layer has the best results in opcode outlier detection and basicblock similarity search. Thus we chose the second-last layer as the output layer in this paper.

Context window for CWP

In this experiment, we evaluate the performance of PalmTree with different context window sizes in the CWP task. For instance, if the context window size is 2, it means that we consider $n-2$, $n-1$, $n+1$ and $n+2$ as contextual instruction when given instruction n as a sample. We evaluate 1, 2, 3, and 4 as four different context window sizes in this experiment. Table 3.11 shows all of the results of the intrinsic metrics when training PalmTree with different context window configurations. We can observe that context window size 1 and 2 have similar performance on the three intrinsic evaluation metrics, but context window size 2 has the best performance on the downstream task EKLAVYA. Further increasing the context window size to 3 and 4 will lead to worse results. Based on these results, we choose the context window size to be 2.

3.4 Related Work

Representation Learning in NLP. Over the past several years, representation learning techniques have made significant impacts in NLP domain. Neural Network Language Model (NNLM) [22] is the first work that used neural networks to model natural language and learn distributed representations for words. In 2013, Mikolov et al. introduced word2vec and proposed Skip-gram and Continuous Bag-Of-Words (CBOW) models [144]. The lim-

Table 3.11: Context Window Sizes

Sizes	**opcode outlier**		**operand outlier**		**bb sim search**	**EKLAVYA**	
	Avg.	Stdev.	Avg.	Stdev.	AUC	Avg.	Stdev.
1	0.864	0.0467	**0.962**	0.0168	**0.923**	0.930	0.0548
2	**0.871**	0.0440	0.944	0.0343	0.922	**0.945**	0.0476
3	0.849	0.0444	0.873	0.0514	0.916	0.908	0.0633
4	0.864	0.0440	0.957	0.0238	0.914	0.916	0.0548

itation of word2vec is that its embedding is frozen once trained, while words might have different meanings in different contexts. To address this issue, Peters et al. introduced ELMo [165], which is a deep bidirectional language model. In this model, word embeddings are generated from the entire input sentence, which means that the embeddings can be dynamically adjusted according to different contextual information.

In 2017, Vaswani et al. introduced transformer [197] to replace the RNN networks (e.g., LSTM). Devlin et al. proposed BERT [50] in 2019, which is a bi-directional transformer encoder. They designed the transformer network using a full connected architecture, so that the model can leverage both forward and backward information. Clark et al. [38] proposed ELECTRA and further improved BERT by using a more sample-efficient pre-training task called *Replaced Token Detection*. This task is an adversarial learning process [70].

Representation Learning for Instructions. Programming languages, including low level assembly instructions, have clear grammar and syntax, thus can be treated as natural language and be processed by NLP models.

Instruction representation plays a significant role in binary analysis tasks. Many techniques have been proposed in previous studies. Instruction2Vec [213] is a manually designed instruction representation approach. InnerEye [224] uses Skip-gram, which is one of the two models of word2vec [144], to encode instructions for code similarity search. Each instruction is treated as a word while a code snippet as a document. Massarelli et al. [141] introduced an approach for function-level representation learning, which also leveraged word2vec to generate instruction embeddings. DeepBindiff [55] also used word2vec to generate representations for instructions with the purpose of matching basic blocks in different binaries. Unlike InnerEye, they used word2vec to learn token embeddings and generate instruction embeddings by concatenating vectors of opcode and operands.

Although word2vec has been widely used in instruction representation learning. It has the following shortcommings: first, using word2vec at the instruction level embedding will lose internal information of instructions; on the other hand, using word2vec at the token level may fail to capture instruction level semantics. Second, the model has to handle the OOV problem. InnerEye [224] and DeepBindiff [55] provided good practices by applying normalization. However, normalization also results in losing some important information. Asm2Vec [52] generates embeddings for instructions and functions simultaneously by using the PV-DM model [115]. Unlike previous word2vec based approaches, Asm2Vec exploits a token level language model for training and did not have the problem of breaking the

boundaries of instructions, which is a problem of token level word2vec models. Coda [65] is a neural program decompiler based on a Tree-LSTM autoencoder network. It is an end-to-end deep learning model which was specifically designed for decompilation. It cannot generate generic representations for instructions, thus cannot meet our goals.

Representation Learning for Programming Languages. NLP techniques are also widely used to learn representations for programming languages. Harer et al. [82] used word2vec to generate token embeddings of C/C++ programs for vulnerability prediction. The generated embeddings are fed into a TextCNN network for classification. Li et al. [121] introduced a bug detection technique using word2vec to learn token (node) embedding from Abstract Syntax Tree (AST). Ben-Nun et al. [19] introduced a new representation learning approach for LLVM IR in 2018. They generated conteXtual Flow Graph (XFG) for this IR, which leverages both data dependency and control flow. Karampatsis et al. [101] proposed a new method to reduce vocabulary size of huge source code dataset. They introduced word splitting, subword splitting with Byte Pair Encoding (BPE) [184] cache, and dynamic adaptation to solve the OOV problem in source code embedding.

3.5 Discussion

In this paper, we focus on training an assembly language model for one instruction set or one architecture. We particularly evaluated x86. The technique described here can be applied to other instruction sets as well, such as ARM and MIPS.

However, in this paper, we do not intend to learn a language model across multiple CPU architectures. Cross-architecture means that semantically similar instructions

from different architectures can be mapped to near regions in the embedded space. Cross-architecture assembly language model can be very useful for cross-architecture vulnerability/bug search. We leave it as a future work.

It is worth noting that instead of feeding a pair of instructions into PalmTree, we can also feed code segment pairs or even basic block and function pairs, which may better capture long-term relations between instructions (currently we use sampling in the context window and data flow graph to capture long-term relations) and has a potential to further improve the performance of PalmTree. We leave this as a future work.

3.6 Conclusion

In this paper, we have summarized the unsolved problems and existing challenges in instruction representation learning. To solve the existing problems and capture the underlying characteristics of instruction, we have proposed a pre-trained assembly language model called PalmTree for generating general-purpose instruction embeddings.

PalmTree can be pre-trained by performing self-supervised training on large-scale unlabeled binary corpora. PalmTree is based on the BERT model but pre-trained with newly designed training tasks exploiting the inherent characteristics of assembly language. More specifically, we have used the following three pre-training tasks to train PalmTree: MLM (Masked Language Model), CWP (Context Window Prediction), and DUP (Def-Use Prediction). We have designed a set of intrinsic and extrinsic evaluations to systematically evaluate PalmTree and other instruction embedding models. Experimental results show that PalmTree has the best performance in intrinsic evaluations compared with

the existing models. In extrinsic evaluations that involve several downstream applications, PalmTree outperforms all the baseline models and also significantly improves downstream applications' performance. We conclude that PalmTree can effectively generate high-quality instruction embedding which is helpful for different downstream binary analysis tasks.

Chapter 4

Evaluating Custom Transformers for Binary Analysis

4.1 Introduction

Inspired by the remarkable progress in large language models (LLMs), also, with the publication of PalmTree, recent research has demonstrated the efficacy of Transformer-based language models [197], language models have gained popularity in the field of binary code analysis. An increasing number of works are now using language models and attempting to customize them to make the models more suitable for specific downstream tasks [216, 10, 119, 202, 159, 96], owing to the shared characteristics of programming languages (PL) including assembly language and natural languages (NL). And this chapter presents a systematic evaluation of our customized assembly language model tailored for these specific tasks.

Almost all these Transformer-based assembly language models (ALMs) have proposed custom pre-training tasks with the purpose of improving the model's understanding on program semantics. For instance, some works [119, 202, 216] employ topological features of binary programs, and design pre-training tasks to capture control flow information, while others [159, 96, 10] aim to capture the operational semantics of assembly code. The majority of them also performed modifications on the model architecture along with their pre-training tasks. For example, jTrans [202] models jump relationships by modifying positional embeddings and predicts jump targets to enable the model to understand jump instructions and the structural connections between basic blocks. StateFormer [159] captures def-use relations and value changes over registers by incorporating new layers of embedding to represent numerical values and applying Neural Arithmetic Unit (NAU) [136] to handle those numerical values.

Despite substantial progress in this area, several questions remain unanswered. First, while existing research papers demonstrate that the proposed architectural modifications are suitable for individual tasks, their generalizability to other tasks remains unclear. For instance, jTrans [202] is designed for function similarity search, and its evaluation was limited to this single task using different baseline models. StateFormer [159] focuses on fine-grained type inference, yet does not assess its approach on other downstream tasks, despite the potential benefits of understanding data changes over execution traces for various binary analysis tasks. An exception is PalmTree [119], which has been evaluated on several downstream tasks, but its focus was solely on instruction-level representation learning, leaving its performance at the function level unexplored. Second, some existing works lack

a systematic and extensive comparison with pre-trained LLMs such as BERT [49] and ALBERT [113]. While jTrans includes a limited ablation study on BERT, it does not provide a comprehensive comparison across all evaluations. Similarly, StateFormer did not compare its model with a pre-trained BERT model, instead evaluating a Transformer model without pre-training as one of its baselines.

To better understand the contributions of these ALMs, we aim to address a fundamental question in this paper: how do these existing designs affect downstream tasks in binary analysis? This question can be broken down into several sub-questions. First, we seek to determine which pre-training tasks are beneficial for multiple downstream tasks. While some pre-training tasks have proven effective for specific downstream tasks, it is unclear whether these tasks can also benefit others or if they might be counterproductive. Second, we aim to evaluate the effectiveness of various architectural modifications.

To address these questions, we conducted multiple evaluations. We selected four ALMs from the binary analysis domain and assessed their performance on four different downstream tasks. Two of these tasks, binary code similarity detection and function type inference, are the original tasks for the models. The third task, algorithm classification, is novel to all the models. The fourth task, Function Name Prediction, was recently introduced by SymLM [96]. Additionally, we applied the pre-training tasks specifically designed for these ALMs to the standard BERT model, which served as our baseline.

From our evaluation results, we have the following observations:

(1) Architectural changes have a limited impact on both pre-training and fine-tuning.

(2) After fine-tuning, the performance gaps between different models are small.

(3) The vanilla BERT models are comparable to or superior to the custom models in the four downstream tasks we evaluated.

Consequently, we conclude that recent architectural modifications to Transformer models, along with tailored pre-training tasks, appear to be unnecessary. Our research suggests that enhancements in fine-tuning techniques might be a more effective way to improve model performance.

We will release the source code of our evaluation framework and related training and testing datasets upon acceptance for publication.

4.2 Evaluation Plan

In this section, we first introduce the models that are evaluated (subsection 4.2.1), evaluation setup (subsection 4.2.2), and data preparation (subsection 4.2.3). Then the evaluations of pre-training tasks are discussed in subsection 4.2.4 and the downstream evaluations are described in subsection 4.2.5.

4.2.1 Models to be Evaluated

Considering the multitude of Transformer-based approaches for various downstream tasks, it is infeasible to evaluate every single one. Therefore, we establish specific criteria for selecting models to be evaluated. First, the pre-trained models must be publicly available, as an official implementation or a pre-trained model shared by the author ensures accurate reproduction of performance. We partially rewrite the source code from certain works with open-source code to match our data format. Second, the papers must

be recently published at premier academic conferences in computer security, software engineering, and machine learning. Third, the approaches must be purely Transformer-based, as our target is to evaluate the customization of Transformer models. Composite models, which require joint training with other models, are beyond our scope. The approaches must include architectural modifications or special pre-training task designs and must be designed for binary code rather than source code or intermediate representation (IR), due to significant differences in semantic structure and preprocessing methods.

Table 4.1: Evaluated Models

Model Name	Architectural Features	Pre-training Tasks	Downstream Tasks
BERT	N/A	MLM	N/A
jTrans	Embedding Layer	MLM, JTP	Function Sim Search
StateFormer	NAU	GSM	Type Inference
Trex	LSTM	MLM, MTP	Function Sim Search
PalmTree	N/A	MLM, CWP, DUP	Intrinsic & Extrinsic

According to previous requirements, We collect models shown in Table 4.1 to perform our evaluation. The source code of these models is publicly available. Furthermore, the dataset of StateFormer is also available for multiple architectures, and the dataset is large-scale. Hence, to simplify our work, we choose to use the pre-trained model provided by the StateFomer and use the dataset to train other models.

In addition to the models mentioned above, we also employed the pre-training tasks proposed by these models to train BERT models. Specifically, we trained BERT-JTP

using the jTrans pre-training task JTP, BERT-GSM using the StateFormer pre-training task GSM, and BERT-CWP and BERT-DUP using the CWP and DUP pre-training tasks proposed in PalmTree [119], respectively. To thoroughly evaluate the performance of these pre-training tasks, we train models of different sizes. This is because some pre-training tasks that may be too hard to train on a standard-sized model might be more feasible to train on a larger model.

4.2.2 Evaluation Setup

We utilized the code provided by the authors of jTrans, StateFormer, Trex, and PalmTree, making necessary modifications to match our data format. Additionally, we implemented the BERT model ourselves as the baseline and pre-trained it on the same configuration for a fair comparison. To make a fair evaluation for all the models, we refer to the original papers of evaluated models and try to apply the most practical hyperparameter configurations for all the standard-sized models. They were trained and fine-tuned with the same number of epochs. We also trained two larger-sized models to validate the effects of customization on larger-scale models, which we refer to as the "L" and "XL" models. Detailed hyperparameter information for these three sizes is provided in Table 5.8.

Table 4.2: Hyperparameters on different sized models

Models	Layers	Dim	# heads	# param	Models	Layers	Dim	# heads	# param
BERT	12	768	12	87M	jTrans	12	768	12	88M
BERT-L	12	1024	16	156M	jTrans-L	12	1024	16	156M
BERT-XL	24	1024	16	307M	jTrans-XL	24	1024	16	308M

Due to the utilization of special tokens for unique pre-training tasks and architecture designs, we cannot use completely identical vocabularies across all models. For instance, jTrans has jump target tokens that share weights with position tokens. StateFormer and Trex have value tokens that are used by the GSM task. Apart from this, we have made every effort to use the same pipeline to ensure that the vocabulary remains as consistent as possible, except for the model-specific special tokens mentioned above.

4.2.3 Data Preparation

We pre-trained all models on the same dataset, which comes from the StateFormer paper. This dataset consists of the latest versions of 33 open-source software projects, including widely used and large projects like OpenSSL, ImageMagick, and Coreutils. We pre-trained the models on x86-64 binaries compiled by GCC-7.5 with four different optimizations (O0-O3), and on three obfuscation strategies (bogus control flow [`bcf`], control flow flattening [`cff`], and instruction substitution [`sub`]), which were implemented using Hikari based on Clang-8. We used Ghidra to disassemble binaries, removed small functions that have less than 10 instructions, and then randomly split the dataset to 80%-20% for training and testing to avoid data contamination. Here, training includes the pre-training of BERT, BERT-JTP, BERT-GSM, BERT-DUP, BERT-CWP and jTrans. It also includes the fine-tuning for any pre-training evaluation and two of the downstream evaluations described in subsection 4.2.4 and subsection 4.2.5. Testing means our evaluation or any validation results we displayed during pre-training and fine-tuning.

For the evaluation of Algorithm Classification, we used a dataset specifically designed for it. More details are included in section 4.2.5. For Function Name Prediction,

due to the need for fine-tuning with specialized labeled data, we performed fine-tuning and evaluation using the dataset provided by SymLM [96]. Specific details can be found in section 4.2.5.

4.2.4 Evaluating Pre-training Tasks

Intuitively, the most straightforward way to assess the effectiveness of a pre-training task is to evaluate the model's performance on this pre-training task directly. This experiment explores the impact of additional pre-training tasks and architectural modifications by comparing the performance of different models on the pre-training tasks.

This research question comprises two sub-questions. Firstly, we investigate whether a language model pre-trained solely through MLM can acquire the same knowledge as models designed for specific tasks and rapidly apply this knowledge through fine-tuning. For example, if a vanilla BERT model, swiftly fine-tuned with a prediction head, can predict jump targets similarly to jTrans, it suggests that JTP pre-training is ineffective.

Secondly, we aim to determine whether architectural modifications introduced alongside pre-training tasks further improve training efficiency. For instance, if a BERT model pre-trained with MLM and JTP achieves performance comparable to jTrans on the JTP task, it indicates that specialized designs like jTrans' embedding layer may be unnecessary.

For these two questions, we will compare three different models: vanilla BERT, BERT with special pre-training tasks (BERT-JTP and BERT-GSM), and customized ALMs (jTrans and Stateformer). To assess whether pre-training tasks and architectural modifications enhance model performance, we connect pre-trained models with an untrained

prediction head and perform supervised fine-tuning. The fine-tuning task is the same as the pre-training task that needs to be evaluated. Since models are pre-trained on the same tasks, they should converge faster and outperform a vanilla BERT model. Below, we outline the two pre-training tasks.

Generative State Modeling

Generative State Modeling (GSM) is a pre-training task proposed by StateFormer [159] to capture value changes involved in arithmetic operations. In this pre-training task, StateFormer is required to predict the values of registers and memories after the execution of each instruction. We consider StateFormer and BERT in this experiment. We try to apply the GSM task on the vanilla BERT model without modifications to the architecture.

StateFormer uses NAU to encode values as input and utilizes a multi-layer Feedforward Network to predict values during pre-training. It minimizes the Mean Squared Error (MSE) between the predicted 8-byte values and the ground-truth 8-byte values for only masked tokens. Note that MSE treats the output byte tokens as numerical values. Since the ground truth should be an integer between 0 and 255, and the loss is a float between 0 and 1, according to the design of the Stateformer [159], we will multiply the predicted result by 256 and round it to calculate the specific predicted value. This is entirely consistent with the evaluation method employed in the original work when probing stateformer on real-world code.

To make the BERT model ready for this evaluation, we generally follow the design of StateFormer and make necessary modifications. We add "mov" instructions before our data sample to initialize registers with input values. For instance, if the register **rax** has

been assigned the value **0xf30f1efa** at the beginning, we put **mov rax, 0xf30f1efa** before the first instruction.

We first pre-train BERT with the default settings. The dataset for pre-training is the same as StateFormer's. Then, we pre-train BERT with MLM and GSM (denoted as BERT-GSM) without modifying the architecture. Moreover, we keep all the constant numbers since the model has to take numerical information to make predictions. Our evaluation is on the byte level with the following formula,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x_i})^2 \tag{4.1}$$

where x_i is the ground truth and $\hat{x_i}$ is the prediction value generated by the model. For accuracy calculation, we transform the output values into integers before making a comparison.

Jump Target Prediction

Jump Target Prediction (JTP) is a pre-training task proposed by jTrans [202] to capture control transfer relations. In JTP, the jTrans model is trained to predict jump targets of jump instructions. We conduct the same experiment to see whether the vanilla BERT model can learn control transfer information without changing the architecture. To accomplish this, we add a fully connected network to the pre-trained language model where the masked jump targets are fed as inputs and the predicted jump locations are generated as outputs.

We mask 70% of the jump targets for training, and compare two training strategies: Using JTP as a fine-tuning task only (BERT), and using JTP in both pre-training (along with the MLM task) and fine-tuning (denoted as BERT-JTP). We use accuracy as the metric in this evaluation and compare BERT-JTP with BERT and jTrans.

4.2.5 Evaluating Downstream Tasks

Our downstream tasks are selected from previous works and include function similarity search, function type inference, and algorithm classification. We chose tasks based on the selected ALMs and their evaluated tasks. For all four tasks, we set BERT, BERT-CWP, BERT-DUP, BERT-JTP, BERT-GSM, jTrans, and StateFormer as our candidate models. BERT-CWP and BERT-DUP are included to further investigate how pre-training tasks designed for instruction embedding influence function-level task performance. Meanwhile, we have also evaluated larger-sized models across all downstream tasks.

To ensure the accuracy of our conclusions, we conducted rigorous statistical tests. We performed t-tests on multiple instances to determine differences between experimental results. Since this work primarily investigates the superiority of customized models over vanilla BERT models, we conducted t-tests between BERT and all other models and calculated their p-values.

Function Similarity Search

Binary function similarity is a building block of many binary security applications such as vulnerability and plagiarism detection. It takes two functions as input and produces a numeric value that represents the similarity between the functions. We conduct this

evaluation to see how different models perform on this well-defined research problem and whether the vanilla BERT model can achieve similar performance. We also follow the function pool evaluation idea of jTrans [202] where each function is compared with every function in the pool. The larger the pool size, the more challenging and realistic this problem becomes.

Let there be a function pool $\mathcal{F}$, and its corresponding ground-truth pool $\mathcal{G}$. For a given query $f \in \mathcal{F}$, we try to find its target ground-truth pair $f^{gt} \in \mathcal{G}$. The retrieval performance can be evaluated using the following two metrics, Where $\mathcal{I}$ denotes an indicator function and is defined as below.

$$Recall@k = \frac{1}{\mathcal{F}} \sum_{f_i \in \mathcal{F}} \mathcal{I}(Rank_{f_i}^{gt} \leq k) \quad \mathcal{I} = \begin{cases} 0, & x = False \\ 1, & x = True \end{cases} \tag{4.2}$$

$$MRR = \frac{1}{\mathcal{F}} \sum_{f_i \in \mathcal{F}} \frac{1}{Rank_{f_i}^{gt}} \tag{4.3}$$

Since our models generate function-level embeddings via Transformer networks and measure similarity using cosine distance, we consider two configurations: one with fine-tuning and one without. Without fine-tuning, we utilize the bare model for generating embeddings and employ cosine distance for measuring their dissimilarity. Conversely, with fine-tuning, we apply contrastive learning to refine the comparison models. More specifically, given a query function f, its ground-truth target f_p, and negative samples $f_{n1}, f_{n2}, ..., f_{ni}$,

$$loss = -log \frac{e^{sim(f,f_p)/\tau}}{\sum_{N}^{i=1} e^{sim(f,f_{ni})/\tau}} \tag{4.4}$$

We divided our dataset into training and testing subsets. Additionally, we set the pool size to 10,000 and then conducted 30 random samplings to obtain multiple values for MRR and Recall.

Type Inference

This downstream task aims to map untyped low-level registers or memory regions, specified by memory offsets, to their corresponding source-level types. We adopt the same experimental design as StateFormer [159]. Specifically, given a sequence of assembly instructions, the model needs to predict the type labels for each operand token in the instructions. It is a classification task, in which some tokens are predicted as the types of function arguments, local, static, or global variables they are associated with, while other tokens do not possess any types. We stack a classification head after different pre-trained models and fine-tune them for type inference. The recovered source-level types can be of different granularities across existing works [27], ranging from primitive types such as int and float to more complex types like struct, array, and recursive types such as trees and lists. We select the most fine-grained type labels from StateFormer [159], which contains 36 different type labels. A detailed list of types can be found in Table 4.3.

As mentioned in StateFormer [159], the dataset for type inference is highly imbalanced, because most of the tokens have the no-access label. Hence, we choose to use the same metrics utilized by StateFormer (i.e., precision, recall, and F1 score) to measure the actual performance. Let TP (True Positive) denote the number of correctly predicted labels, FP (False Positive) denote the wrong ones, TN (True Negative) denote no-access

Table 4.3: The types that are predicted as output

Type	**Name**
Placeholder	no-access
Primitive	int, unsigned int, long, unsigned long, long long, unsigned long long, short, unsigned short, char, unsigned char, float, double, long double
Aggregate	struct, union, enum, array
Pointer	Aggregate *, Primitive *, void *

tokens which have been correctly predicted and FN (False Negative) denotes tokens with other types being predicted as no-access. And we have $Precesion = TP/(TP + FP)$, $Recall = TP/(TP + FN)$, $F1 = 2 * \frac{Precision*Recall}{Precision+Recall}$. We also conducted 10 random samplings of the test set and repeated the experiments multiple times to avoid the randomness of the results.

Algorithm Classification

This downstream task aims to differentiate algorithms used in different binaries. In this task, the model needs to classify the assembly code according to its functionality. Some works [150, 131] treat this task as a code clone detection task, which is similar to the Function Similarity Search described in the previous sections.

We use the POJ-104 dataset [150] for this task. The POJ-104 dataset originates from a pedagogical programming open judge (OJ) system [150] that automates the eval-

Table 4.4: The difference between Datasets

Dataset	#Functions per binary	Binary Sizes	#Classes	#Functions per class
POJ104	1-2	∼50KB	50	∼500
Function Sim Search	more than 10	100KB-10MB	pool size	less than 10

uation of submitted source code for specific problems by executing the code. As depicted in Table 4.4, the POJ-104 dataset significantly differs from the one utilized in Function Similarity Search. Moreover, the fine-tuning scale is much smaller compared to Function Similarity Search, which poses a greater challenge for the models to capture and learn the features of this dataset effectively. In essence, this task resembles few-shot learning for the models. Consequently, if a model gains more advantages from the customization of architectures and the pre-training tasks, it is expected to exhibit more pronounced benefits in the obtained results.

To maintain as much similarity as possible with the configuration used for downstream evaluation, we also employed different optimization levels and three obfuscation strategies. We trained and tested the model using a total of **56,439** binaries.

This task aims to retrieve R targets for a given binary from the fine-tuning/testing sets, with the Mean Average Precision (MAP) as the evaluation metric, where R is the number of other binaries in the same class. Each data sample is labeled with one of 104 programming problems (50 of which compile correctly). Some source files contain multiple functions, which we address by concatenating all assembly code and removing compiler-added helper functions. We employ 10-fold cross-validation, splitting the dataset by class to avoid randomization bias. Training involves 40 classes (with 10 for validation), while

testing uses the remaining 10. The average training set comprises approximately 17,500 binaries, with the testing set containing around 4,200 binaries.

We use the Mean Average Precision (MAP) as the evaluation metric. $MAP = \frac{1}{M}\sum_{m=1}^{M} AP(m)$ Where M is the number of query functions, $AP(m)$ is the average precision score when having query function m. We prepared two evaluations for this downstream task, with and without fine-tuning. We use the same fine-tuning process proposed by [131]. Still, we use the same models as Function Similarity Search in this evaluation.

Function Name Prediction

This downstream task aims to predict function names in stripped binaries. In this task, the model needs to predict the name of a given function based on its semantics. Given a function f, we define the function name prediction task as a multi-class and multi-label classification problem. In detail, we first encode a function f using any transformer model and generate a function embedding $\mathcal{E}$. Then, we aim to train a decoding function $\mathcal{R}$ that maps $\mathcal{E}$ to a function name set $\mathcal{W}$, which consists of a set of tokens $\mathcal{W} = t_1, t_2, ..., t_i$. Here, the function tokens t_i can represent common English words, programmers' commonly used abbreviations, numbers, and so on. $\mathcal{W}$ belongs to a function name vocabulary $\mathcal{V}(\mathcal{V} \supseteq \mathcal{W})$. And we have $\mathcal{W} = \mathcal{R}(\mathcal{E})$

In this evaluation, we utilize the framework of SymLM [96], which provides an open-source implementation. Furthermore, SymLM is implemented using the open-source pre-trained model from Trex [158], which is also one of our evaluation targets. However, we faced challenges in reproducing their fine-tuning process due to the absence of a publicly available dataset (with only a dataset generation tool being provided by the author).

Additionally, the fine-tuning process proved to be excessively time-consuming, taking approximately 8 days to complete the fine-tuning of SymLM with the Trex model and an MLP decoder. These limitations hindered our ability to replicate their experimental setup precisely. Hence, we utilized the x86 dataset released along with the code of SymLM, which has 43,436 function samples for training, 5,043 for validation, and 10,954 for testing with mixed optimization levels. To evaluate different models, we choose to use the same metrics as in Type Inference (precision, recall, and F1 score). More specifically, given the ground truth function name set $W = \{w_1, w_2, w_3, ..., w_n\}$, and predicted function name $\hat{W} = \{\hat{w_1}, \hat{w_2}, \hat{w_3}, ..., \hat{w_m}\}$, they define a membership function:

$$\mathbb{1}(W, \hat{w}) = \begin{cases} 1, & \hat{w} \in W \\ 0, & \hat{w} \notin W \end{cases} \tag{4.5}$$

which indicates whether the predicted token $\hat{w_m}$ is in the ground truth set W. Based on this indicator function, we then calculate the true positive, false positive, and false negative:

$$tp = \sum_{\hat{w_i} \in \hat{W}} \mathbb{1}(W, \hat{w}), fp = \left\|\hat{W}\right\| - tp, fn = \|W\| - tp, \tag{4.6}$$

where the $\|\bullet\|$ denotes the number of tokens in the name set. Subsequently, we get precision, recall and F1-score using the formula described in section section 4.2.5. Similar to the previous downstream evaluations, we also sampled the testing set 10 times and obtained multiple results to mitigate the randomness.

4.3 Evaluation Results

4.3.1 Pre-training Tasks

Generative State Modeling

Table 4.5 shows the results of the Generative State Modeling task. We noticed that the dataset contains a significant number of 0 values (attributable to the small values of many constant numbers, which result in zero-padding in the high digits). Thus, we introduce "Accuracy w/o 0" to evaluate the model's accuracy on predicting non-zero values. We also put an accuracy curve during training in the appendix.

The experimental results show that the prediction accuracy is quite low, which is expected given the complexity of modeling data changes in assembly code through a regression task. BERT-GSM shows faster learning in the early stages, but all models exhibit high instability with fluctuating accuracy during training. Additionally, BERT-XL does not perform better in this task, likely due to the inherent difficulty of GSM for language models.

Jump Target Prediction

As described in section 4.2.4, we choose BERT, BERT-JTP, and jTrans to evaluate the Jump Target Prediction task. Table 4.5 presents the evaluation results.

We observe that after fine-tuning, jTrans only slightly outperforms BERT and BERT-JTP. Analysis of the training process reveals that jTrans initially trains faster than other models but is quickly caught up by BERT and BERT-JTP. Notably, the larger BERT-XL consistently outperforms jTrans-XL.

Table 4.5: Results of Pre-training Tasks

Generative State Modeling			Jump Target Prediction	
Model	Acc	Acc w/o 0	Model	Acc
BERT	0.141	0.063	BERT	0.780
BERT-GSM	0.179	**0.092**	BERT-JTP	0.797
BERT-XL	0.148	0.056	BERT-XL	**0.903**
Stateformer	0.129	0.053	jTrans	0.822
			jTrans-XL	0.756

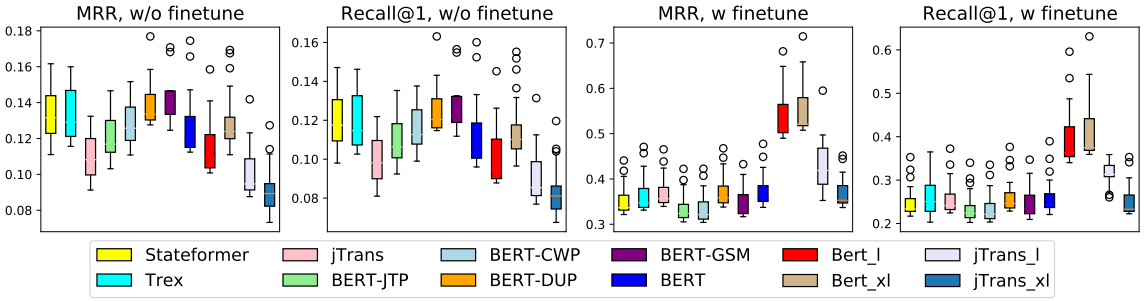


Figure 4.1: MRR/Recall@1 on Function Similarity Search. Pool size = 10000.

Our evaluation of the pre-training task indicates that modifications to the model architecture do not significantly improve the performance of the associated pre-training tasks. For overly challenging Generative State Modeling, even the new component NAU does not aid the model in learning the associated pre-training task better.

4.3.2 Downstream Tasks

Function Similarity Search

The results of the function similarity search experiments are presented in Figure 4.1. Before fine-tuning, the MRR and Recall of all the models are below 0.20. Among these, the vanilla BERT's performance is similar to most models, while jTrans, jTrans-L, and jTrans-XL exhibit significantly lower performance than BERT in terms of MRR and recall@1 (p-value ≤ 0.05). However, after fine-tuning, the performance of large-scale models (BERT-L and BERT-XL) significantly outperforms other models, while the remaining models demonstrate similar performance. It is worth noting that no model exhibits a significant advantage over the vanilla BERT model. jTrans does not outperform the BERT model as originally reported [202], because the dominant influence is the application of contrastive learning during the fine-tuning process. Based on this evaluation, we can conclude that neither architectural changes nor custom pre-training tasks introduce any tangible benefits. More advanced contrastive learning has a dominant effect.

Type Inference

Table 4.6 presents the results of the evaluated models on type inference. Precision, recall, and F1-score are averaged from multiple samplings, with p-values calculated from statistical analysis between F1 scores of other models and BERT. Notably, except for jTrans-XL, the remaining models perform similarly, with BERT-L achieving the best performance.

This suggests that fine-tuned models do not show significant performance differences due to pre-training and architecture variations. However, jTrans-XL stands out, where its customized embedding layers negatively impact performance.

Table 4.6: Results of Type Inference (Opt-level=Mixed)

Model	Precision	Recall	F1-score	P-value
BERT	0.903	0.904	0.904	-
BERT-JTP	0.887	0.888	0.888	3.9×10^{-9}
BERT-CWP	0.888	0.888	0.888	6.9×10^{-9}
BERT-DUP	0.901	0.900	0.901	2.8×10^{-1}
BERT-GSM	0.901	0.902	0.902	6.2×10^{-3}
BERT-XL	0.897	0.889	0.893	2.0×10^{-5}
BERT-L	**0.936**	**0.935**	**0.936**	6.9×10^{-18}
StateFormer	0.889	0.892	0.890	2.4×10^{-40}
Trex	0.870	0.875	0.872	7.5×10^{-38}
jTrans	0.907	0.908	0.908	2.5×10^{-8}
jTrans-L	0.912	0.913	0.913	6.2×10^{-10}
jTrans-XL	0.501	0.254	0.338	3.1×10^{-39}

We can also see that additional pre-training tasks for BERT do not make any benefits for this downstream application. In fact, these pre-training tasks appear to have confused the BERT model, causing degradation to the performance of this downstream task.

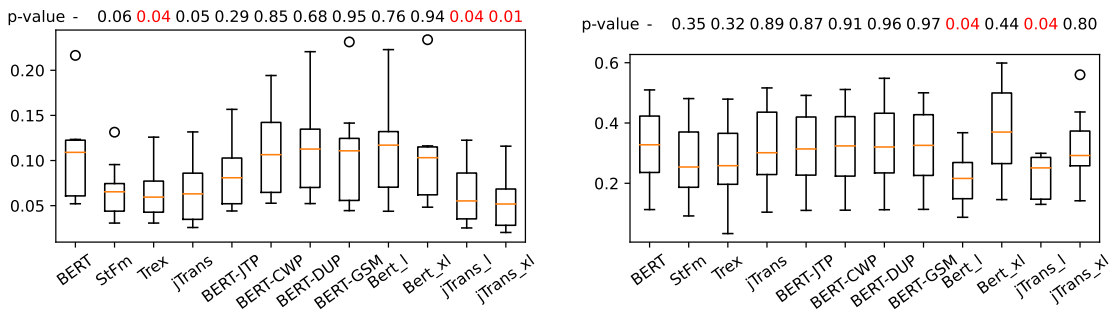


Figure 4.2: Results of Algorithm Classification with (top) and without fine-tuining.

Based on these observations, we are confident to conclude that the vanilla BERT model, paying no effort on architectural modifications and extra costs for additional pre-training tasks, is the most suitable and cost-effective choice for this task. On the contrary, the specifically designed model, StateFormer, is actually incapable of improving its own targeted task. It is worth noting that the authors of StateFormer did not conduct a head-to-head comparison between StateFormer and Vanilla BERT but instead performed an ablation study. Since StateFormer does not include the MLM task during its pre-training phase, no model in the ablation study is equivalent to Vanilla BERT. Trex, which shares a similar design, also fails to surpass the performance of BERT.

Algorithm Classification

Figure 4.2 lists the results of the evaluation on algorithm classification. The p-value shows the T-test results between the target model and BERT. Here, StFm denotes the StateFormer. We observe that without fine-tuning, BERT is among the best. However,

the differences between BERT and other models are not statistically significant, except for Trex, jTrans, and jTrans-XL (i.e., with a p-value less than 0.05). After fine-tuning, the differences between different models are even smaller, with even less statistical significance, meaning the p-values are generally larger. In general, all models perform poorly on this task. Therefore, we can see that customizations do not provide any benefits in this task.

Function Name Prediction

Table 4.7 shows the results of function name prediction. Similar to section 4.2.5, precision, recall, and F1-score here represent the mean results obtained from multiple samplings of the test set. The p-value is derived from the t-test between the F1-scores of other models and those of the BERT model. We can see that larger models generally outperform standard-sized models, with BERT-XL achieving the best F1 score. Among models of the same size, all models exhibit similar performances. No model demonstrates a considerably higher F1-score compared to Vanilla BERT.

Results of the Function Similarity Search task signify that the advanced contrastive learning technique is very effective, whereas specially designed pre-training tasks and architectural changes fail to introduce any tangible benefits. The vanilla BERT model is comparable or superior to the specifically modified architectures in the Type Inference, Algorithm Classification, and Function Name Prediction.

Table 4.7: Results of Function Name Prediction

Model	Precision	Recall	F1-score	P-value
BERT	0.749	0.440	0.554	-
BERT-JTP	0.781	0.398	0.528	6.09×10^{-21}
BERT-CWP	0.750	0.440	0.554	9.15×10^{-1}
BERT-DUP	0.799	0.416	0.547	4.90×10^{-11}
BERT-GSM	0.794	0.407	0.538	3.72×10^{-18}
BERT-L	**0.812**	0.501	0.619	7.24×10^{-17}
BERT-XL	0.807	**0.503**	**0.619**	3.62×10^{-20}
Stateformer	0.711	0.497	0.585	8.42×10^{-20}
Trex	0.711	0.496	0.584	7.28×10^{-19}
jTrans	0.760	0.452	0.567	4.43×10^{-17}
jTrans-L	0.798	0.466	0.588	1.32×10^{-19}
jTrans-XL	0.796	0.459	0.582	5.82×10^{-17}

4.4 Discussion

This research endeavors to evaluate the efficacy of existing pre-training tasks and architectural changes for Transformer-based assembly language models. Nevertheless, it is crucial to clarify that our conclusions do *not* negate the potential importance of all architectural changes. Our evaluation indicates that the architectural modifications in Stateformer, Trex, and jTrans do not provide sufficient advantages in downstream tasks to justify the

costs associated with modifying the model architecture. we did not evaluate other architectural changes proposed in other works due to limited access to their models. However, our study sends a signal to the research community that architectural changes proposed in future works must undertake a rigorous evaluation.

Likewise, our evaluation does *not* establish that the introduction of new pre-training tasks is completely ineffective for ALMs. On the contrary, we posit that the level of difficulty in pre-training tasks for ALMs and the interplay between different pre-training tasks are crucial factors that limit the performance of ALMs. Based on the existing experimental findings, a good pre-training task should have an appropriate level of training difficulty that constantly challenges the language model throughout the training process and forces it to learn the desired features. Additionally, this task should not interfere with other pre-training tasks. Besides, further investigation and experimentation are needed to fully explore the potential of novel pre-training tasks in advancing the capabilities of ALMs.

Finally, our study highlights fine-tuning as the most straightforward and effective technique, assuming readily available well-labeled datasets for training and testing, which is often feasible in many binary analysis tasks. For example, diverse sets of binaries can be generated by enumerating different compilers and options, with ground truth obtainable from debug symbols. Consequently, fine-tuning proves highly effective. However, it is crucial to recognize that sparse or low-quality labeled datasets for a specific downstream binary analysis task may lead to different conclusions.

4.4.1 Our Suggestions

Based on the experimental findings and the aforementioned discussions, we offer the following recommendations for future research:

- Always take the vanilla BERT or other vanilla transformer-based models into consideration or provide a comprehensive ablation study.
- When proposing a specialized model for a specific downstream task, provide a rationale for the model's suitability solely to this task or to conduct evaluations towards multiple tasks.
- Try to improve training efficiency, such as applying contrastive learning, before customizing the model.

4.5 Related Work

Deep Learning Models in Program Analysis Tasks.. Program analysis is a long-studied research area. In recent years, Transformer-based pre-trained language models have been widely applied to numerous binary and source code analysis tasks, and our work only covers a subset of these tasks. In the task of function similarity search [216, 74, 158, 10, 219], apart from models we evaluate, there exist proprietary models that are not included in our evaluation. OrderMatters [216] uses BERT to model sequential instruction sets from basic blocks, and CNN to model the topological features. It concatenates the representations and uses an MLP layer to generate embeddings for functions. COMBO [219] utilizes not only assembly code but also source code and related comments for similarity search. Bin-

Bert [10] and Trex [158] benefit from dynamic information. BinBert [10] combines assembly instructions with symbolic expressions and uses the BERT model to encode the concatenated inputs. This execution-aware Transformer model is proven to be able to benefit the binary understanding. Trex [158] adopts an approach that is highly similar to StateFormer, so it is not reevaluated in this work.

Transformer-based models are also utilized in various other tasks. XLIR [75] uses the Transformer-based model to match binaries and source code at the intermediate representation (IR) level. SymLM [96] focused on function name recovery. This approach jointly models the execution behavior of the calling context and instructions with the comprehensive function semantics via a Transformer-based encoder. BinProv [85] uses BERT to generate embeddings for provenance classification. VulHawk [133] uses a graph neural network along with a Transformer language model to identify vulnerabilities across architectures. Like COMBO [219] and OrderMatters [216], VulHawk [133] also tries to merge different kinds of features including imported functions, strings, and control-flow graphs.

There are many approaches that utilize other deep learning models to solve program analysis problems. Several works [208, 55, 122, 216, 133, 217, 221, 140, 104] try to use Graph Neural Network (GNN) to capture structural features of functions, while SAFE [141] and InnerEye [223] using LSTM with attention mechanism to encode assembly code. The GNN is usually used to encode control flow graphs [208, 55, 122, 216]. It has also been used in disassembly. DeepDi [215] constructs a graph model called Instruction Flow Graph to capture different instruction relations and use a Relational Graph Convolutional Network (RGCN) to propagate instruction embeddings for accurate instruction classification.

However, these models fall outside the scope of this work, as they integrate other approaches, such as Graph Neural Networks, which are not categorized as language models.

Evaluations on Neural Binary Analysis Approaches.. Benchmarks play an important role in deep learning-based program analysis research. CodeXGLUE [131] provides a benchmark for code intelligence problems including clone detection, Defect Detection, Cloze test, Code completion, Text-to-code generation, etc. The work encourages the development of language models that have the capability to address a wide range of program understanding and generation problems, with the goal of increasing the productivity of software developers. We share the same viewpoint on this matter. However, this paper focuses only on source code-based approaches and downstream tasks. The conclusions and insights derived from this study cannot be directly applied to binary analysis evaluations, as well as the metrics employed.

PalmTree [119] introduced an evaluation framework for instruction embeddings, using intrinsic and extrinsic metrics, but it focused on instruction-level models, generating embeddings for each instruction. In contrast, our work targets function-level embeddings. The intrinsic evaluations by PalmTree may not align with the goals of function-level models. PalmTree concluded that control-flow and data-flow information can help instruction representation learning. However, recent studies [216, 202, 158, 159, 96] have shown that learning from longer sequences is more effective. Our evaluation also shows that PalmTree's pre-training tasks do not outperform the vanilla BERT model with function-level sequences. Marcelli et al. [139] re-implemented and evaluated existing works on function similarity search, finding that many recent papers show similar accuracy levels when

evaluated on the same dataset, despite claiming state-of-the-art advancements. Our evaluation reaches a similar conclusion. While Marcelli et al. focus on head-to-head comparisons of a single downstream task, our work assesses the generalizability of models across multiple downstream tasks.

4.6 Conclusion

In this work, we have evaluated custom Transformer-based models and their specifically designed pre-training tasks by collecting, tailoring, implementing, and testing four recent models including jTrans, PalmTree, StateFormer, and Trex, together with tailored pre-training tasks. We have evaluated the vanilla BERT model and these models with four major downstream tasks: Function Similarity Search, Type Inference, Algorithm Classification, and Function Name Prediction. According to our evaluation, we have observed that: certain pre-training tasks (e.g. GSM) are too challenging for the Transformer model to learn effectively; Architectural changes do not bring tangible benefits for both pre-training and fine-tuning. Moreover, improving fine-tuning (e.g., contrastive learning for Function Similarity Search) is generally more beneficial than introducing new pre-training tasks or making architectural modifications.

In light of our findings, our more comprehensive evaluation has revealed some potential issues with recent modifications to model architectures and newly introduced pre-training tasks. Our research indicates that the key to improving the performance of Transformer-based models in downstream tasks lies primarily in fine-tuning. Other architecture changes and pre-training changes must be justified.

Chapter 5

Learning an Assembly Language Model for Zero-Shot Obfuscation Detection

5.1 Introduction

Building on the previous discussion, we recognize that vanilla language models, when appropriately fine-tuned, can perform on par with or even surpass custom models across various downstream tasks. However, our subsequent research highlights challenges with supervised fine-tuning for certain specialized tasks, such as code obfuscation.

Code obfuscation is a technique used to deliberately make source code, binary code, or program logic difficult to understand, interpret, or reverse-engineer. It alters the code's structure and syntax without changing its functionality. The main objectives are to

protect intellectual property, prevent unauthorized access, and hinder reverse engineering by reducing code readability and analysis. However, obfuscation is also commonly exploited for illegal purposes, including malware development [212], code plagiarism [107], and intellectual property theft [205]. Detecting code obfuscation helps security professionals identify hidden or malicious behaviors, making it a crucial element of modern cybersecurity strategies.

Obfuscation techniques are mainly divided into data obfuscation, static code rewriting, and dynamic code rewriting [181]. Obfuscation detection for binaries mainly targets the latter two, especially static code rewriting. Security researchers initially used certain statistical features, such as entropy [134] and n-grams [100], to detect code obfuscation. However, these approaches often only work well against specific obfuscation techniques. Some studies have also attempted to use machine learning techniques [191, 24, 178, 195, 72], such as Naïve Bayes (NB), k-Nearest Neighbor (KNN), and Decision Tree (DT), to detect obfuscated code.

In recent years, various deep learning models have been widely applied to the field of binary static analysis at an astonishing speed and have quickly achieved state-of-the-art performance in various tasks [210]. As one of the most important tasks at the forefront of the reverse engineering workflow, obfuscation detection has also greatly benefited from the application of deep learning. Researchers have attempted to use Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) to encode assembly code [220, 194], along with word2vec [143, 145] word embedding models, achieving better performance compared to traditional machine learning methods.

Many learning-based approaches to obfuscation detection frame the problem as a supervised classification task, relying on known obfuscation methods for training. While this allows these models to perform well on familiar obfuscation techniques, it limits their ability to generalize to novel or proprietary methods. Commercial software vendors and malware authors, seeking to protect their binaries from reverse engineering, often employ non-public, custom obfuscators. This lack of sample diversity in supervised learning can introduce data bias, causing models to overlook features that, while insignificant in the training set, may be critical for detecting previously unseen obfuscation techniques. Consequently, existing learning-based obfuscation detection methods may face similar limitations, raising concerns about their generalizability despite strong performance on standard evaluations. Indeed, our evaluation in subsection 5.3.6 shows that supervised learning methods have very low detection rates (~ 0.04) for realworld malware samples, which are often protected by custom obfuscators.

In this work, we address the challenge of detecting code obfuscation from the perspective of assembly language modeling. Although both regular and obfuscated binary code "speak" the same assembly language (adhering to its syntax and grammar), how they speak differs significantly. We believe that the assembly language produced by regular binary code is more straightforward, concise, and comprehensible, as it originates from source code written by human developers following sound software engineering practices and is compiled to maximize efficiency. In contrast, obfuscated binary code tends to "speak" assembly in a more convoluted manner, deliberately designed to obscure its logic and hinder analysis by human experts and reverse engineering tools.

Therefore, we propose to train an assembly language model, which can capture how regular binary code "speaks" the assembly language. After training, this assembly language model can detect obfuscated binary code, because its style significantly deviates from that of the regular binary code in training. Specifically, we train a Transformer-based language model using the causal language modeling (CLM) task on a large corpus of regular binary code produced by different compilers and compiler options. Consequently, this model captures the linguistic style of regular binary code. We then detect if a given binary code is obfuscated by measuring how accurate this model predicts the next token in its assembly code sequence. A common metric for this prediction is called "perplexity". So when the perplexity of a given binary code exceeds a predetermined threshold, this input binary is deemed obfuscated. Evidently, the proposed obfuscation detector is a zero-shot detector and is capable of detecting obfuscated code produced by custom and previously-unseen obfuscators, because it is only trained on regular/unobfuscated code.

To further separate obfuscated code apart from regular code, we propose two new metrics: Error Perplexity (EP) and Consecutive Error Perplexity (CEP). The error perplexity metric sets focus on error predictions only, whereas consecutive error perplexity further stresses on a sequence of mis-predicted tokens. Our evaluation shows that these new metrics indeed improve the detection accuracy.

To evaluate the efficacy of the proposed approach, we implement a prototype called ALMOND [1]. Specifically, we train a GPT-1.0 model on unobfuscated assembly code with a large dataset with 5200 ELF binaries and 3000 PE binaries over a wide variety of compilers and configurations, using the causal language modeling (CLM) task. Using our newly

[1]ALMOND stands for **A**ssembly **L**anguage **M**odel for **O**bfuscatio**N** **D**etection.

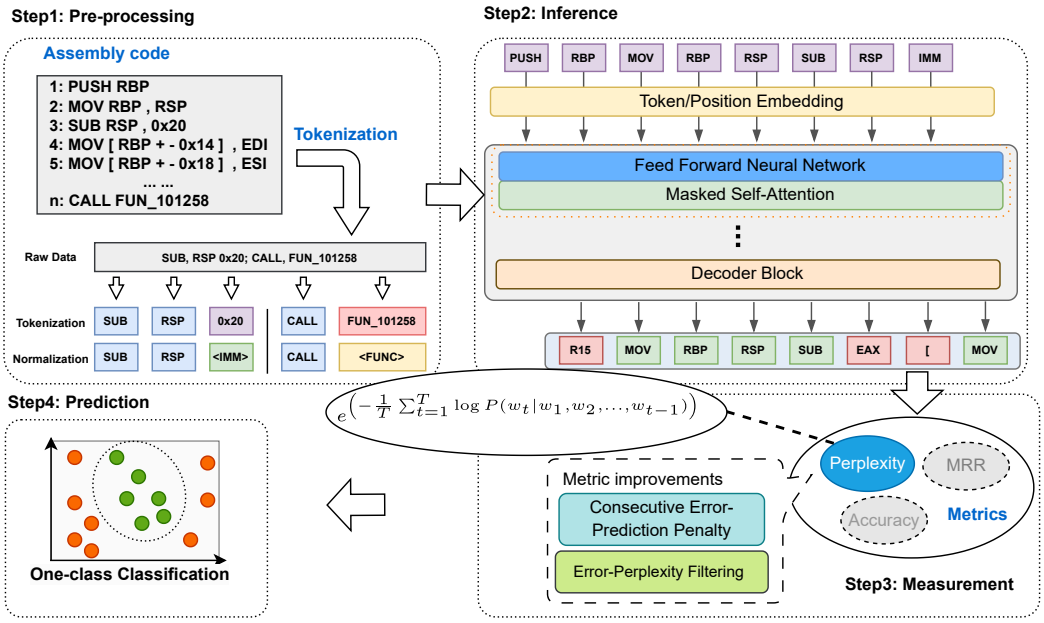


Figure 5.1: The Overview of ALMOND

proposed CEP metric, ALMOND has an accuracy of 96.3% on binaries with unseen obfuscation methods which is much higher than machine learning and deep learning approaches, and is also superior to the fine-tuned language model. In real-world evaluation, ALMOND significantly outperforms the supervised fine-tuned language model ALMOND-S, demonstrating its effectiveness with a 0.869 AUC score which significantly outperforms fine-tuned language models. Our evaluations demonstrate that, although supervised learning-based models can achieve excellent results on experimental datasets, in complex and unknown real-world environments, an unsupervised, 0-shot model like ALMOND proves to be more reliable.

5.2 Design

To meet the challenges, we propose ALMOND, a novel zero-shot approach for detecting code obfuscation in binary executables, which does not rely on labeled training data or supervised learning. Figure 5.1 illustrates the pipeline of ALMOND. We start by training a language model using unobfuscated assembly code. Once training is complete, we can directly use the model for anomaly detection on obfuscated code. We predict the input tokens using the language model's pre-training task, evaluate the prediction results using metrics like error-perplexity, and classify them based on a set threshold. In the following sections, we will provide a detailed explanation of the design for each step. In subsection 5.2.1 and subsection 5.2.2, we will first introduce the preprocessing and pre-training processes. In subsection 5.2.3, we will focus on how we achieve zero-shot detection by reusing the pre-training task and utilizing the newly proposed error-perplexity metric and the consecutive error-prediction penalty operator.

5.2.1 Pre-processing

In the pre-processing stage, we disassemble the binary and tokenize the assembly code. Natural language models require a tokenizer to convert raw text into numerical vectors, and tokenization methods generally fall into two categories: word-based and subword-based. As the name implies, word-based tokenization treats each word as a separate token, using spaces as delimiters along with some auxiliary rules. While this approach is simple, it results in a large vocabulary size and introduces the issue of out-of-vocabulary (OOV) words. For instance, "dog" and "dogs" would be considered entirely different tokens in a word-based tokenizer.

To address the OOV problem, modern NLP models typically use subword-based tokenizers, such as Byte Pair Encoding (BPE) [66] or WordPiece [206]. BPE iteratively merges the most frequent pairs of bytes or characters until the target vocabulary size is reached. Similarly, WordPiece constructs subwords iteratively by selecting token sequences that maximize the likelihood of the text, based on subword frequency data learned during training.

However, assembly language differs significantly from natural languages in terms of structure, syntax, and vocabulary. Assembly code has a more rigid structure and a much smaller vocabulary. For example, in x86-64 assembly, there are only around 1,000 unique mnemonics and 100 registers and symbols. As a result, word-based tokenizers do not encounter the same challenges as they do with natural languages. In this context, subword-based tokenizers offer little advantage. On the contrary, word-based tokenizers are more efficient due to their simplicity and lack of training requirements. Consequently, previous research [119, 202, 52] on assembly language models has commonly adopted the approach of separating opcodes and operands based on spaces.

The application of word-based tokenization to assembly code is not without challenges. Assembly code contains many immediate values and addresses, which can still lead to significant out-of-vocabulary (OOV) issues. Moreover, these tokens vary across different binaries, even when compiled from the same source code, due to variations in compilers and platforms. Immediate values and addresses will differ accordingly. Training a model to predict these specific values reduces its accuracy on non-obfuscated code and increases perplexity, thereby impairing the model's ability to detect obfuscated code. This issue is

present for both subword-based and word-based tokenizers. To address this problem, we implemented token normalization [52, 119]. Specifically, as shown in Figure 5.1, we replace immediate numbers and string tokens within instructions with special tokens. This allows the model to focus on the underlying semantics without being influenced by specific numerical or address information, which are often subject to configuration changes. These normalized tokens make it easier for the model to learn and make accurate predictions.

5.2.2 Architecture

For our language model architecture, we choose to utilize state-of-the-art Transformer-based models. There are three main types of popular Transformer architectures: pure Encoder models (e.g., BERT [49]), also known as auto-encoding Transformers; pure Decoder models (e.g., GPT [170]), also known as auto-regressive Transformers; and Encoder-Decoder models [204], which combine elements of both.

Encoder-only models, such as BERT, utilize a bi-directional attention mechanism and are trained through Masked Language Modeling (MLM). However, this approach is not well-suited for ALMOND. The obfuscation detection task is more akin to a stylometric analysis problem, where the goal is to differentiate between the language styles of typical compilers and obfuscators. BERT, being trained on MLM tasks, focuses primarily on predicting masked tokens by leveraging the full context. As a result, it emphasizes semantic and syntactic understanding, with little sensitivity to variations in language style. This makes BERT less effective for obfuscation detection, as the model is likely to predict masked tokens accurately, regardless of whether the code has been obfuscated, as long as it understands the syntax and semantics of the input context.

In contrast, pure Decoder models like GPT use only the Decoder module of the Transformer architecture. At each step, the attention layer can access only the preceding words in the sequence, enabling the model to iteratively predict subsequent words based on the context already generated. This approach is known as Causal Language Modeling (CLM). When predicting the next tokens, the model must consider both fine-grained syntax and semantics, as well as generate sequences that match the style of the preceding text. As a result, if the GPT model has been pre-trained predominantly on unobfuscated code, it will face greater difficulty in predicting sequences for obfuscated code, which exhibits a distinct language style.

One of our experiments confirmed this hypothesis. Table 5.1 presents the accuracy of GPT's CLM task and BERT's MLM task on both obfuscated and unobfuscated code (on validation Dataset during pre-training). The results show that for the MLM task, the top-1 prediction accuracy and perplexity are similar for both types of sequences. This indicates that BERT struggles to distinguish between the two styles. Consequently, for ALMOND, GPT and the CLM approach are more appropriate design choices.

Table 5.1: Accuracy(Top-1) and Perplexity on BERT and GPT

Model	Obfuscated Code	Unobfuscated Code
BERT(MLM) Accuracy	0.877	0.895
GPT-1.0(CLM) Accuracy	0.725	0.856
BERT(MLM) perplexity	2.728	2.014
GPT-1.0(CLM) perplexity	4.045	2.225

We only use unobfuscated assembly code to train the GPT model. As previously mentioned, this allows our GPT model to learn only the language style of unobfuscated code, which will be used for subsequent obfuscation detection. We train the GPT model using a causal language modeling (CLM) task. More specifically, Transformer architecture is used to model the conditional probabilities $P(w_t \mid w_1, w_2, \ldots, w_{t-1})$ The model is trained to predict the next word in a sequence, given the previous words. The training objective is

$$\text{Loss} = -\sum_{t=1}^{T} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1}) \tag{5.1}$$

5.2.3 0-Shot Obfuscation Detection

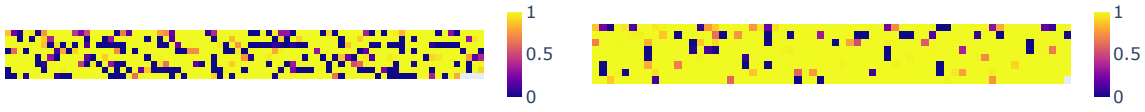
After training, the pre-training task will be reused to perform obfuscation detection. When a query code snippet is fed into the GPT model, it will make predictions from w_2 to w_n if the input length is n. Although obfuscated code functions the same as unobfuscated code, obfuscated instruction sequences create significant logical differences. For a language model trained on unobfuscated code, predicting the logic of obfuscated code becomes challenging. Therefore, in theory, we can evaluate the model's predictions using various metrics designed to assess language model predictions, and then classify the code by setting a threshold. A common example of such a metric is perplexity. For a particular token in a sequence, perplexity is calculated as:

$$\text{Perplexity}(w_t) = \exp\left(-\log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{5.2}$$

For a sequence, perplexity is calculated as:

$$\text{Perplexity}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{5.3}$$

Here, we can see that perplexity is essentially the exponential of the average loss per token. If the perplexity exceeds the threshold, it indicates poor prediction results, leading us to classify the input sample as obfuscated code.



(a) Probability of an obfuscated binary (b) Probability of a regular binary

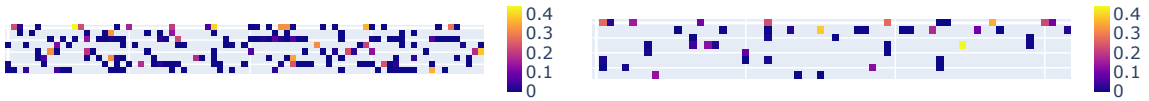
Figure 5.2: Comparison of probability between obfuscated and regular binaries

5.2.4 Further improvement on Obfuscation Detection

Table 5.1 shows that the perplexity predicted by the GPT model already exhibits a significant difference between obfuscated and unobfuscated code. Therefore, perplexity is an appropriate metric for zero-shot obfuscation detection. However, we found that there is still room for improvement. The example in Figure 5.2 drove us to further investigate. The figure shows the prediction probability on ground truth tokens of an obfuscated code snippet, where each square represents the GPT model's probability of predicting the ground truth token at a specific position. Lighter colors indicate higher probabilities, while darker colors not only reflect lower probabilities but also signal potentially incorrect predictions. Compared to unobfuscated code in Figure 5.2(b), some interesting features were observed.

First, it is evident that for both obfuscated and unobfuscated code, the prediction probability for most tokens is quite high, as supported by the data in Table 5.1. Both

obfuscated and unobfuscated code achieves over 70% accuracy. Our further tests reveal that the perplexity of these correctly predicted tokens is very similar, as shown in Table 5.2. Therefore, we can infer that these correctly predicted tokens do not significantly contribute to obfuscation detection. However, the small subset of incorrectly predicted tokens plays a crucial role, as low probabilities result in high perplexity.



(a) Probability of an obfuscated binary (b) Probability of a regular binary

Figure 5.3: Comparison of probability with mispredictions only

Figure 5.3 presents a heatmap after masking all the correct predictions. It can be observed that, although the number of dark squares in the obfuscated code is higher than in the unobfuscated code, the unobfuscated code also contains many dark squares in terms of color. However, the dark squares in the obfuscated code exhibit a clear consecutiveness, while those in the unobfuscated code are more scattered and discrete(Only horizontally connected tokens represent consecutive tokens.).

Error-perplexity

Based on the previous observation, we propose error-perplexity as the metric for classification. Instead of using the perplexity of all tokens, we only consider the perplexity of incorrectly predicted tokens as the evaluation factor. As mentioned earlier, in obfuscated

Table 5.2: Perplexity on correct predictions

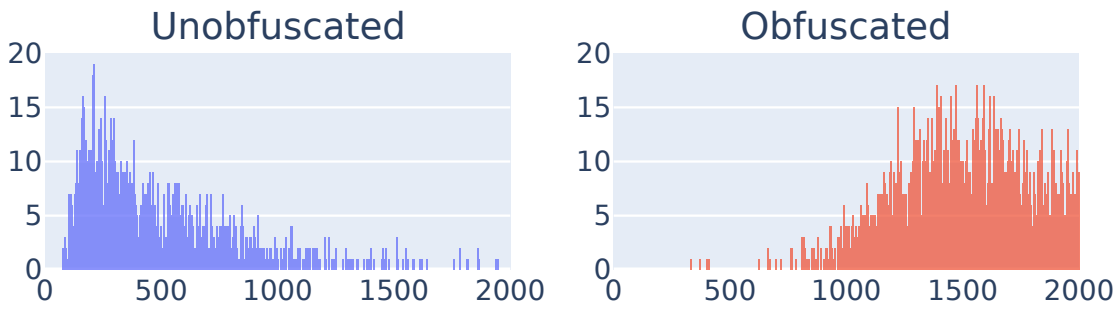
Code	Mean	Max	Min
Regular	1.12	4.00	1.00
Obfuscated	1.28	4.14	1.00

code, many tokens can still be predicted by the GPT model, and for these tokens, the perplexity will be low regardless of whether the code is obfuscated, introducing noise. However, for tokens that the GPT model predicts incorrectly, obfuscated and unobfuscated codes fall into different scenarios. For non-obfuscated code, incorrect predictions for a token are often due to the presence of multiple possibilities within normal logic. For example, after a test instruction, various jump instructions may reasonably follow, leading to potential errors in prediction. As a result, the ground truth token is typically among these possible tokens, leading to a relatively low perplexity value. On the other hand, incorrect predictions are more frequent in obfuscated code than in non-obfuscated code. These errors often arise because the language model cannot predict the obfuscated code's unique logic based on the previous tokens. In such cases, the predictions tend to be more random, and the ground truth token's probability is very low, resulting in a significantly higher perplexity value. Error-perplexity uses Equation 5.4 as follows.

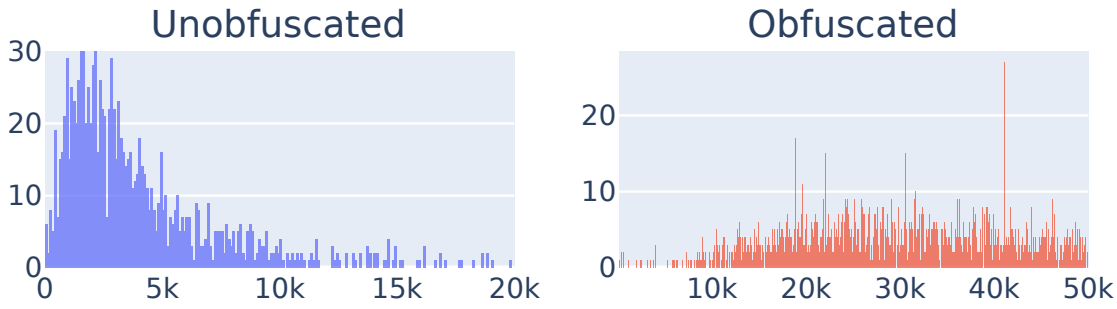
$$\text{Error-Perplexity}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|M|}\sum_{t \in M} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \quad (5.4)$$

Where:

- M is the set of indices where the model made an incorrect prediction.
- $|M|$ is the size of the set M, i.e., the number of mispredicted tokens.



(a) Distributions of error-perplexity



(b) Distributions of error-perplexity with CEP

Figure 5.4: Comparison of Distributions of Error-Perplexity with and without CEP

We collected the distribution of error-perplexity for both unobfuscated and obfuscated code, which can be found in Figure 5.4(a). It can be observed that the obfuscated code is primarily distributed in the region above 1000, while the unobfuscated code is concentrated in the range between 1 and 1000.

Table 5.3: Avg. length of error predictions

Code	Regular	Obfuscated
Avg. length of error predictions	1.519	2.335

Consecutive Error Perplexity

On top of error-perplexity, we introduced a new mechanism called Consecutive Error prediction (CEP). Our investigation into language model predictions highlights two key scenarios where prediction errors arise. First, the semantics are correct, but the model faces multiple valid choices. In this case, the model usually predicts the opcode correctly and can often predict the operands as well. Even if it fails to predict the operands, the perplexity remains relatively low. Second, when an obfuscator rewrites a sequence of instructions rarely seen in regular binaries, the language model tends to make errors in both the opcodes and operands, and sometimes even in subsequent instructions. As a result, the occurrence of consecutive prediction errors is significantly higher in obfuscated code than in regular code. In Table 5.3, we present the average number of consecutive token prediction errors for both obfuscated and unobfuscated code (This means that a single token prediction error has a length of 1, two consecutive token prediction errors have a length of 2, and so on, with the average being taken.). It can be observed that the average length of prediction errors for obfuscated code exceeds 2.

Therefore, when calculating error-perplexity, we introduce the Consecutive Error Prediction (CEP) mechanism. This method is inspired by the joint probability of indepen-

dent events. Since the GPT model already has access to the entire target sequence during training, it can compute predictions for all positions at once. This is because the training data includes both the full input and target sequences (by shifting the target sequence to the right by one position, multiple training samples are created). As a result, the model's prediction for each position is treated as an independent event. Therefore, for an input sequence that has consecutively mispredicted tokens, the CEP is defined as follows:

$$\text{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{S_i \in \mathcal{S}} \log P(S_i \mid w_1, \ldots, w_{t_1-1})\right) \tag{5.5}$$

Where:

- $\mathcal{S}$ is the set of sequences of consecutive mispredicted tokens.
- $|\mathcal{S}|$ is the number of sequences in $\mathcal{S}$.
- $P(S_i \mid w_1, \ldots, w_{t_1-1})$ represents the joint probability of the sequence S_i, conditioned on the preceding tokens $w_1, w_2, \ldots, w_{t_1-1}$.

Then $P(S_i \mid w_1, \ldots, w_{t_1-1})$ on the sequence $S_i = (w_{t_1}, w_{t_2}, \ldots, w_{|S_i|})$ can be expanded as:

$$P(S_i \mid w_1, \ldots, w_{t_1-1}) = \prod_{j=1}^{|S_i|} P(w_{t_j} \mid w_1, w_2, \ldots, w_{t_j-1})$$

Substituting this expanded form into the original CEP equation, we get:

$$\text{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{S_i \in \mathcal{S}} \log \prod_{j=1}^{|S_i|} P(w_{t_j} \mid w_1, w_2, \ldots, w_{t_j-1})\right)$$

Using the logarithmic property $\log \prod = \sum \log$, the equation simplifies to:

$$\text{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{S_i \in \mathcal{S}}\sum_{j=1}^{|S_i|}\log P(w_{t_j} \mid w_1, w_2, \ldots, w_{t_j-1})\right) \tag{5.6}$$

Where:

- $\mathcal{S}$ is the set of sequences of consecutive mispredicted tokens.
- $|\mathcal{S}|$ is the number of sequences in $\mathcal{S}$.

By simplifying the summary process, the formula becomes:

$$\text{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{t \in m}\log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{5.7}$$

According to Equation 5.7, it can be observed that the value of CEP depends on the value of $|\mathcal{S}|$. With a fixed number of incorrect predictions, if the consecutive incorrect predictions increase, the number of error sequences $|\mathcal{S}|$ decreases, leading to a higher CEP. Furthermore, compared to Equation 5.4, We can observe that, for the same sequence, the value of error-perplexity is the simple average of all individual predictions. CEP changes based on the number of consecutive error sequences. If all error predictions are consecutive, then $|\mathcal{S}| = 1$ and CEP reaches its maximum. Conversely, if all error predictions are non-consecutive, CEP reduces to error-perplexity.

5.3 Evaluation

In this evaluation, we aim to answer the following research questions:

1. **RQ1:** How does ALMOND's performance compare to that of a supervised fine-tuned classifier when applied to known obfuscation methods?

2. **RQ2:** How does ALMOND perform compared to a supervised fine-tuned classifier on previously unseen obfuscation methods?
3. **RQ3:** How does ALMOND perform under different configurations (e.g., metrics and model sizes)?
4. **RQ4:** How does ALMOND perform on real-world cases?

5.3.1 Implementation

We employed the GPT-1.0 as our model architecture, which is considered small by contemporary standards. It consists of 12 transformer layers, each with 12 heads. It has an output dimension of 768 and an intermediate layer dimension of 3072. We implemented the GPT model using Hugging Face's framework and conducted pre-training on a server with a single A100 40GB GPU.

5.3.2 Dataset Collection

Obfuscators. We collect four obfuscators for evaluation: OLLVM [99][2], Hikari [3], Tigress [4], and Alcatraz [5]. OLLVM, a modification of LLVM, integrates obfuscation into the compilation process and provides three main obfuscation algorithms: Instructions Substitution, Control Flow Flattening, and Bogus Control Flow. Hikari builds upon OLLVM, offering five additional obfuscation methods: AntiClassDump, FunctionCallObfuscate, FunctionWrapper, IndirectBranching, and StringEncryption. Tigress, in contrast, is a source-to-source

[2]https://github.com/obfuscator-llvm/obfuscator
[3]https://github.com/HikariObfuscator/Hikari
[4]https://tigress.wtf/index.html
[5]https://github.com/weak1337/Alcatraz

Table 5.4: Obfuscators and Transformation Methods

Obfuscators	Transformations
OLLVM	Instruction Substitution, Bogus Control Flow Control, Flow Flattening
Hikari	Anti-Class Dump, Function Wrapper, Function Call Obfuscate Indirect Branching, String Encryption
Tigress	Add Opaque, Flatten Functions Split Fucntions, Merge Functions
Alcatraz	Obfuscation of Immediate Moves, Control Flow Flattening, ADD Mutation, Lea obfuscation

transformer designed for the C language. Unlike OLLVM and Hikari, which operate during compilation, Tigress takes a C source program as input and outputs an obfuscated C program. For this evaluation, we selected the AddOpaque, Split, Merge, and Flatten obfuscation techniques from Tigress to obfuscate the source code and then compiled it into binary form.

It is important to note that we used the O0 optimization level during compilation for both OLLVM and Tigress. For source-to-source obfuscators like Tigress, subsequent compiler optimizations could remove or reduce the effectiveness of the obfuscation techniques. Thus, using the O0 optimization level ensures that the original obfuscation algorithms are preserved as much as possible. Table 5.4 summarizes the obfuscators and the respective transformations used in our training and testing datasets.

Alcatraz represents obfuscators that directly modify binaries. This tool works on x64 PE binaries, which is the only platform supported by Alcatraz. It provides powerful obfuscation features including obfuscation of immediate moves, control flow flattening, ADD mutation, and LEA obfuscation. Since it targets PE binaries and is not derived from OLLVM, its implementation differs significantly from OLLVM and Hikari, which were used in pre-training. Alcatraz also includes many dynamic encryption features such as entry point obfuscation and anti-disassembly. However, since these two obfuscation techniques are outside the scope of our research, we modified the Alcatraz source code and recompiled it to disable these methods.

Pre-training Data. In evaluation of StateFormer [159], the authors collected 33 open-source projects in their latest versions, including well-known and large projects such as OpenSSL, ImageMagic, and Coreutils. These projects were compiled for four instruction set architectures including x86, x64, MIPS, and ARM, each with four different optimizations using GCC-7.5. We used the x64 portion of the StateFormer training set. Additionally, the Stateformer dataset includes obfuscated code generated using Hikari and OLLVM, we did not use these obfuscated binaries to pre-train ALMOND.

It is worth noting that our language-based baseline models also used a portion of this dataset set for fine-tuning. To further enhance data diversity, we additionally collected 3,000 PE binaries from Windows systems for pre-training, with the goal of increasing the variety of binaries across different platforms and compilers.

Testing Data. For testing, we selected binaries that were entirely distinct from those used in pre-training. Specifically, we used the POJ-104 dataset, which originates from a

Table 5.5: Performance on known obfuscation methods

Model	Accuracy	Precision	Recall	F1
Naïve Bayes	0.942	0.911	0.975	0.958
BiGRU-CNN	0.933	0.936	0.932	0.934
ALMOND-S	**0.988**	**0.988**	**0.984**	**0.985**
ALMOND	0.962	0.953	0.975	0.963

pedagogical programming open judge (OJ) system[150] designed to automate the evaluation of submitted source code for specific problems. For our test set, we compiled over 14,000 POJ-104 binaries, encompassing more than 25,000 functions.

Real-world dataset. For evaluations on real-world cases, we collect 5000 Real-world binaries from Linux Distributions and Windows PE files. Similarly, we selected 5000 binaries labeled as malware from VirusTotal feedings. These binaries were compiled for different platforms using various compilers, and the malware likely employed various obfuscators or packers, resulting in significant diversity.

5.3.3 RQ1: How does ALMOND perform on known obfuscation methods?

In this experiment, we investigate the performance gap between ALMOND and supervised classifiers when dealing with known obfuscation methods. To provide a basis for comparison, we established the following baselines. First, we replicated the method proposed by Salem et al. [179] as the baseline for traditional machine learning. This method

encodes assembly code using TF-IDF [189] and utilizes Multinomial Naïve Bayes (MNB) as the classifier. MNB is a variant of the Naïve Bayes algorithm specifically designed for classification tasks involving discrete features and is commonly used in text classification problems [102]. At the same time, it can also be effectively combined with TF-IDF. We refer to this approach as Naïve Bayes.

Second, we implemented Tian et. al.'s OBOB [194] as the deep learning baseline. In this approach, code sequences are sampled from the control flow graph using a shortest path algorithm. A Bi-directional GRU network is then used to encode the sequences, followed by a CNN for further dimensionality reduction. Finally, classification is performed using a softmax classifier. For clarity, we refer to this model as BiGRU-CNN.

Finally, we utilized the pre-trained ALMOND model, attached it to a classifier with a Multi-Layer Perceptron (MLP) network [176], and conducted fine-tuning. This model is referred to as ALMOND-S, in which S stands for **S**upervised Learning. For ALMOND, we determined the optimal threshold using a set of labeled data from the Stateformer dataset (which contains unobfuscated code and obfuscated code with Hikari and OLLVM), and we applied a fixed threshold value for the evaluation of RQ1 and RQ2. However, different training sets may result in different thresholds. Due to ALMOND's flexibility, the threshold can be adjusted at any time based on the data.

It is worth noting that there is no overlap between the obfuscated binaries used during training and fine-tuning and those used in the test set during evaluation.

Table 5.5 presents the results of different models on a test set containing known obfuscation methods. We observe that, after 18 hours of fine-tuning, ALMOND-S is the

Table 5.6: Performance on unseen obfuscation methods.

Models	Tigress				Alcatraz			
	A	P	R	F1	A	P	R	F1
Naïve Bayes	0.563	0.943	0.103	0.185	0.522	0.945	0.106	0.190
BiGRU-CNN	0.862	0.866	0.862	0.863	0.566	0.884	0.247	0.247
ALMOND-S	**0.966**	**0.966**	0.957	**0.961**	0.622	0.793	0.452	0.576
ALMOND	0.963	0.952	**0.969**	**0.961**	**0.967**	**0.964**	**0.958**	**0.961**

best performer, achieving an accuracy of 0.988 and an F1-score of 0.985. Notably, without any fine-tuning or supervision, ALMOND achieves an accuracy of 0.962 and an F1-score of 0.963, demonstrating that in a zero-shot setting, ALMOND still delivers performance comparable to fine-tuned models with the same architecture.

Although Naïve Bayes and BiGRU-CNN perform slightly worse than ALMOND, they still achieve accuracy and F1-scores above 0.93, indicating that both supervised learning and zero-shot learning approaches can effectively detect known obfuscation methods.

5.3.4 RQ2: How does ALMOND perform on previously unseen obfuscation methods?

This experiment examines the performance gap between ALMOND and supervised classifiers when dealing with unseen obfuscation methods. To ensure precise control over the unseen dataset, we trained Naïve Bayes, BiGRU-CNN, and fine-tuned the language model using only the binaries obfuscated with OLLVM. For testing, we used binaries

obfuscated with Tigress and Alcatraz, ensuring that the obfuscation methods employed in Tigress were not present in the OLLVM-based training data.

The experimental results are presented in Table 5.6.BiGRU-CNN and ALMOND-S demonstrated relatively better generalizability on Tigress binaries. However, there was a significant performance drop on Alcatraz binaries. ALMOND-S achieved the best results among the baselines, showcasing the advantage of language models in semantic modeling.

In contrast, ALMOND remained unaffected when applied to previously unseen obfuscation methods and significantly outperformed all other baselines. For binary-based obfuscation methods with completely different implementations, ALMOND showed higher perplexity and reduced token prediction accuracy. In comparison, supervised methods often classified these obfuscated binaries as benign, resulting in false negatives. This also confirms our earlier assertion that, in an environment with unknown obfuscation methods, the generalizability of unsupervised training is superior to that of supervised training.

5.3.5 RQ3: How does ALMOND perform under different configurations?

In this experiment, we examined how different metrics and model sizes affect the performance of ALMOND.

Performance of Different Metrics

Table 5.7 presents the results of five different metrics applied to both unobfuscated and obfuscated binaries, using various obfuscation methods. The goal is to highlight the numerical differences between regular and obfuscated binaries. The results show that, for

Table 5.7: Performance of Different Metrics

Method	Accuracy	MRR	Perplexity	Error-Perplexity	CEP
regular	0.750	0.836	9.434	570.481	4.53k
sub	0.715	0.801	22.657	2002.313	15.37k
fla	0.811	0.859	11.002	1801.224	24.28k
bcf	0.647	0.724	25.615	1816.941	22.25k
tigress	0.623	0.812	18.494	2295.242	38.80k

accuracy (Acc) and Mean Reciprocal Rank (MRR), there is no significant difference between unobfuscated and obfuscated binaries. In some cases, binaries obfuscated with control flow flattening (FLA) even achieve higher accuracy and MRR than their unobfuscated counterparts. This suggests that using accuracy or MRR as a classification metric could result in false negatives, where obfuscated binaries—particularly those obfuscated using control flow flattening—are incorrectly classified as unobfuscated.

In contrast, for perplexity-based metrics (i.e., original perplexity (PPL), error perplexity (EP), and consecutive error perplexity (CEP)), we can observe wide numeric gaps between regular and obfuscated binaries.

To further evaluate how these metrics affect ALMOND's performance, we plot their ROC curves in Figure 5.5 and compute the Area Under the Curve (AUC). We can observe that the original perplexity achieves reasonable performance with an AUC of 0.947, whereas error perplexity outperforms it by a big margin with an AUC of 0.978, and consecutive error perplexity further beats error perplexity with an AUC of 0.992.

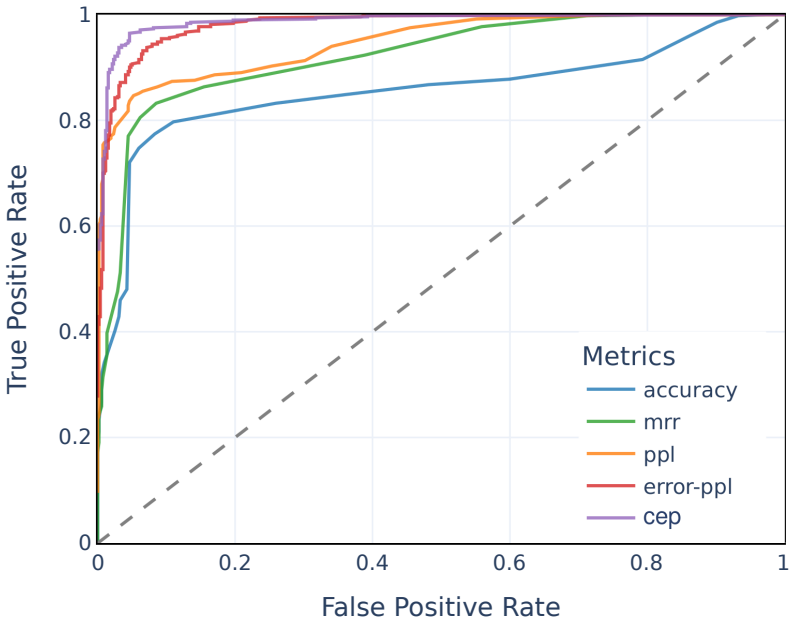


Figure 5.5: ROC curves on different metrics

Table 5.8: Hyperparameters on different sized models

Models	Layers	Dimensions	Heads	Parameters
Small	6	768	4	14M
Standard	12	768	12	87M
Large	24	768	12	172M

Table 5.9: Performance on different model sizes

Models	Recall	Precision	F1-score	AUC
Small	0.944	0.927	0.935	0.982
Standard	0.963	0.958	0.960	0.991
Large	0.939	0.894	0.915	0.977

In summary, this experiment demonstrates that both the proposed error-perplexity and the Consecutive Error Perplexity (CEP) significantly enhance the performance of ALMOND.

Performance of Different Model Sizes

In this subsection, we will examine the model's performance under different sizes. We adjusted the model size by varying the number of layers and self-attention heads. Including the default size, we trained a total of three different model sizes, labeled as ALMOND-small, ALMOND, and ALMOND-large. Table 5.8 provides detailed parameters for each of these models.

Table 5.9 shows the precision, recall F1-score and AUC scores of ALMOND on different sizes. We observed that in the zero-shot setting, a larger model size does not always lead to better performance; instead, there is a sweet spot. As we scale from ALMOND-small to ALMOND, performance improves with the increase in layers. However, further increasing the model size results in a performance drop. This is because our model relies on its understanding of the semantics of unobfuscated code to distinguish obfuscated code.

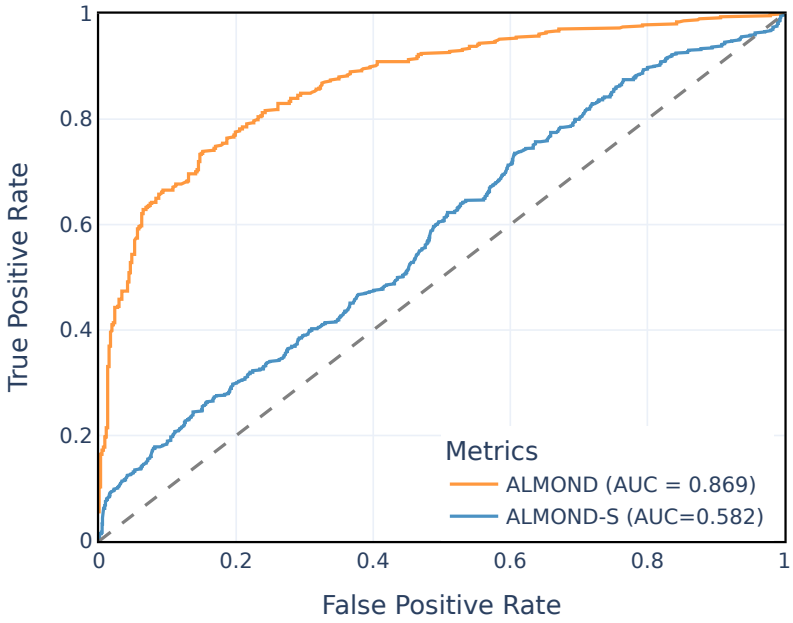


Figure 5.6: ROC curves on real-world binaries

If the model is too small, it lacks the capacity to fully grasp the semantics of assembly code, leading to excessively high perplexity values on unobfuscated code. Conversely, if the model is too large, its understanding of the assembly code semantics becomes too strong, and its generalization ability too high. As a result, it predicts low perplexity for obfuscated code. Therefore, there is a sweet spot in the model size of ALMOND, where it can adequately understand the semantics of assembly code without losing its ability to distinguish obfuscated code through error-perplexity due to overgeneralization.

5.3.6 RQ4: How does ALMOND perform on real-world cases

This experiment evaluates the performance of ALMOND on large-scale, real-world programs, including both commercial off-the-shelf (COTS) and open-source software.

In practice, binaries are typically unobfuscated, so even a low false positive rate can undermine ALMOND's effectiveness due to the high volume of binaries. Additionally, real-world binaries are compiled with various compilers and configurations and may include scientific computation libraries. These libraries often contain extensive mathematical operations that can resemble the logic used by obfuscators. As a result, such operations could be mistakenly identified as obfuscated, leading to false positives. The experiment aims to determine ALMOND's applicability in a real-world malware detection environment. The purpose of the experiment is not to suggest that we should use ALMOND alone for malware detection. Instead, it offers a new perspective on zero-day detection. This is based on a previously discussed assumption that malware will inevitably use some form of obfuscator or packer to obfuscate the source code in order to evade anti-virus detection. Hence, ALMOND can be employed to identify potential zero-day malware, serving as an initial filter. Once an alert is triggered, slower but more comprehensive program analysis tools and reverse engineering techniques can be used to investigate and confirm the nature of the suspected malware.

Unlike the previous experiments, this experiment requires classification at the binary level, so we adjusted the calculation method for the metrics accordingly. Instead of calculating CEP at the function level, we now compute CEP for the entire binary. Therefore, in this experiment, we plotted ROC curves using different thresholds.

Figure 5.6 shows ALMOND's performance on real-world binaries. We observed that ALMOND achieved an AUC of 0.869 and an F-1 score of 0.803 on this dataset, indicating that ALMOND can also achieve high accuracy in real-world malware detection tasks. Overall, as an initial filter in a comprehensive malware detection system, ALMOND is fully capable of fulfilling this role.

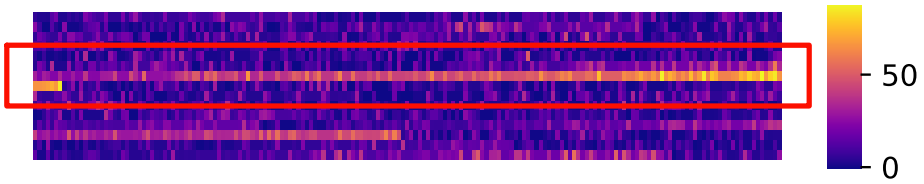


Figure 5.7: The heatmap of malware A

On the contrary, the classification accuracy of ALMOND-S dropped significantly. As a classifier utilizing a language model, ALMOND-S possesses strong language modeling and feature extraction capabilities, as demonstrated in RQ1. However, as a supervised model, ALMOND-S is unable to handle such tasks in real-world environments.

Real case: Packers

Case 1 Figure 5.7 presents a heatmap of the perplexity with CEP on a malware sample. In the heatmap, some distinct highlighted lines can be observed. We extracted one segment in Figure 5.8 for analysis. This segment originates from the function at address 0x40161b. This function appears to be part of an encryption/decryption or decompression routine, containing complex operations such as multiplications, bit shifts and rotations, and bitwise comparisons along with conditional jumps. It also makes use of bitwise instructions like LZCNT, SWAP, NEG, etc. It is known to all, that some packers typically decrypt or unpack compressed executable data on the fly, so we believe this function might be responsible for part of that process—transforming the packed data into executable code just before it is executed. It is worth mentioning that we also studied the implementation of encryption algorithms in OpenSSL. However, we found that the encryption algorithms

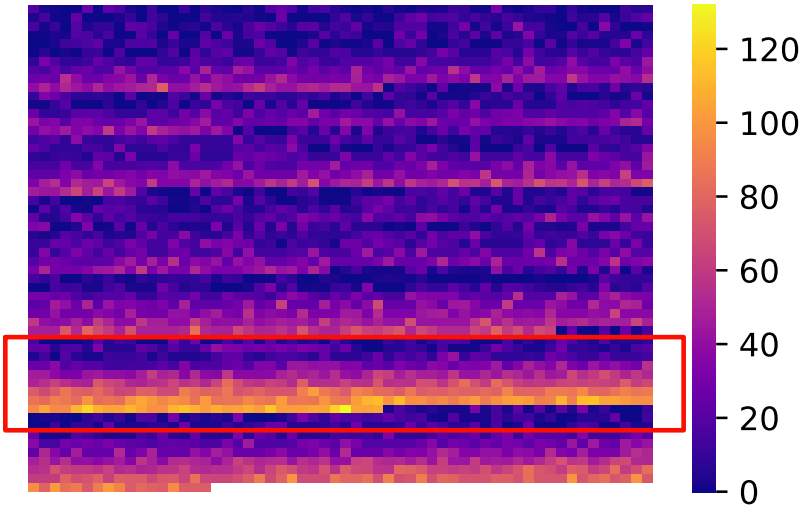


Figure 5.9: The heatmap of malware B

used in network communications are not the same as the function in this case. Taking the x86_aes_encrypt function as an example, this function primarily uses a combination of MOVZX, XOR, SHR, and similar instructions, mainly performing logical and bitshift operations. However, the case function mainly involves various specialized instructions with arithmetic and bit manipulation. Since OpenSSL-related functions are included in ALMOND's pre-training dataset, the language model can also accurately predict these operations.

Furthermore, we believe this segment was either manually crafted by the author or generated using a specialized obfuscation tool, which is completely different from the obfuscated segments observed in other experiments. Hence, ALMOND made consecutive prediction errors in this segment. Besides this, we also observed a few similar segments in the binary. For the rest of the program, ALMOND exhibited lower perplexity.

Figure 5.8: Assembly code snippet of malware A

```
IMUL        EAX, EBX
BSF         EBX, param_2
LZCNT       EAX, EBX
BSWAP       EAX
CMP         AL, param_2
IMUL        EAX, EBX
NEG         AL
MOVSX       EAX, param_2
SAR         AL, 0x67
CMP         AL, param_2
SWAP        EAX
XCHG        BL, AL
BSWAP       EAX
JMP         0x401668
```

Case 2 Figure 5.9 presents the heatmap of another malware sample. Information from VirusTotal indicates that this program likely used some VM-based techniques to protect and encrypt its logic. However, there are still some functions that can be fully disassembled. The heatmap reveals that one part of the binary has higher perplexity compared to other sections. Upon further investigation, this function appears to be a part of a process injection routine, leveraging various Windows API functions to inject code (likely a DLL or shellcode) into another process and execute it remotely. It achieves this by creating a new process or accessing an existing one, allocating memory in that process, writing the payload to the allocated memory, and finally creating a remote thread to execute the injected payload. ALMOND exhibited clear anomalies when dealing with process injection operations, as such operations are rarely found in typical programs. This resulted in very high perplexity and consecutive prediction errors during inference.

5.3.7 Efficiency

In this experiment, we evaluated the efficiency of ALMOND. We found that on an A100 40G GPU, with a batch size of 32, ALMOND can achieve a throughput of

173.808 inferences or 88989 tokens per second, and with a batch size of 64, it can achieve a throughput of 220.324 inferences or 112640 tokens per second.

5.4 Related Works

5.4.1 Obfuscation Detection

As discussed in subsection 2.3.2, prior research on obfuscation detection primarily aimed to facilitate reverse engineering, employing rule-based, machine learning-based, and deep learning-based methods. The approach proposed in this paper builds on deep learning methods while also leveraging metrics used in rule-based detection. Previous deep learning approaches, such as those by Zhao et al. [220] and Tian et al. [194], utilized CNNs and LSTMs for supervised learning. However, with the application of language models in binary analysis, the use of language models and the pre-training, fine-tuning paradigm has become a superior solution. To our knowledge, this paper is the first to use a language model for obfuscation detection. The connections and distinctions between this work and other studies that apply language models to static binary analysis will be discussed in. After modeling assembly language with a language model, this paper introduces a novel language model-based metric, error perplexity, to detect obfuscated code. This approach is analogous to using rule-based metrics like entropy [134] or n-gram models [100] for obfuscation detection, but with a key difference: the proposed metric is designed to assess the predictions of the language model rather than directly targeting assembly code or raw bytes.

5.4.2 Language Model for Static Binary Analysis

In recent years, numerous studies have explored the use of language models for static binary analysis. PalmTree [119], for example, proposed using language models to generate instruction embeddings, which can be applied to various downstream tasks. Most of these studies have focused on leveraging language models by introducing specialized pre-training tasks or innovative model architectures to target specific tasks, such as similarity detection [202, 158, 126, 95, 6, 141, 216], type inference [159], function name recovery [96, 17], and value set analysis [79]. These tasks are typically accomplished through fine-tuning or by introducing special pre-training tasks.

The key distinction between this work and those studies is that our approach does not involve any additional pre-training tasks or fine-tuning. Instead, it relies solely on the default pre-training of GPT, enabling obfuscation detection to be performed in a zero-shot manner.

5.4.3 Zero-shot Classification and Anomaly Detection

Zero-shot classification involves predicting a class the model has never encountered during training, often requiring it to perform tasks not explicitly learned. A notable example is GPT-2, tested on downstream tasks like machine translation without fine-tuning [172]. In this context, the model classifies input text into unseen labels. Two primary approaches exist: Puri et al.[167] utilize generative capabilities of models like GPT by prompting the model with task descriptions and candidate labels, while Zhang et al. [218] map both labels and documents into a high-dimensional space to predict labels using cosine similarity. Unlike

Zhang et al. [218], ALMOND focuses on One-Class Classification, avoiding the need for label mapping. Similarly, in anomaly detection (AD), the task is to identify data instances that deviate from the norm [175], with applications in security, such as detecting DDoS attacks or monitoring system logs [91, 90, 117, 54]. Zero-shot anomaly detection employs two approaches: one utilizes large-scale data and powerful models through meta-learning or in-context learning, as seen with Liu et al. [127] and RAGLog [156]. The second, more computationally efficient approach studies sample features and applies scoring methods for evaluation, enabling zero-shot detection in tasks like pixel-level anomaly detection [118, 183, 53]. This feature analysis method is the focus of this paper.

5.5 Discussion

This paper proposes a zero-shot obfuscation detection method based on a pre-trained language model, achieving results comparable to fine-tuned models with significantly less training data. However, there are limitations to this work. Firstly, the paper only explores binary classification using error-perplexity. In reality, the information predicted by the language model could be used for more detailed classification tasks, such as identifying specific obfuscation methods, all without requiring fine-tuning. We believe that building on this work, incorporating few-shot learning algorithms such as Generalized Learning Vector Quantization [180] could enable the prediction of obfuscation methods.

Secondly, this paper presents only a prototype of classification based on error-perplexity and does not thoroughly investigate combining multiple metrics to further enhance performance. However, we have already observed that combining perplexity and error-perplexity outperforms using either metric alone.

Lastly, this paper does not include experiments to thoroughly examine the potential vulnerabilities of ALMOND to evasion or adversarial attacks, which will be discussed here. First, adversarial learning methods targeting binary classifiers must ensure that the modified binary can both mislead the machine learning classifier and maintain functional integrity [188]. Consequently, these methods typically avoid altering the original assembly code and instead insert data or code between the assembly instructions to deceive the classifier. However, the classification method based on error-perplexity proposed in this paper focuses solely on the language model's incorrect predictions. As a result, for an obfuscated binary, it would be challenging to deceive ALMOND by simply inserting unobfuscated code. However, this does not imply that ALMOND is entirely immune to evasion. Attackers would need to design obfuscation methods that more closely resemble regular code to reduce the error-perplexity value, which inherently means a reduction in the effectiveness of the obfuscation itself. Therefore, we have reason to believe that ALMOND offers stronger resistance to adversarial attacks compared to other supervised-learning-based approaches.

5.6 Conclusion

We present ALMOND, a zero-shot obfuscation detector based on a transformer language model. We employed a metric-based classification technique along with an anomaly

detection approach and proposed the error-perplexity and Continuous Error-prediction Penalty to further enhance detection capabilities. Our evaluation demonstrates that ALMOND achieves an accuracy of 96.3% on binaries with previously unseen obfuscation methods, surpassing traditional machine learning and deep learning approaches. Moreover, in a real-world malware detection task, ALMOND achieved a false negative rate of less than 0.001, while maintaining a false positive rate of just 0.269. ALMOND has proven that obfuscation detection can be achieved in a zero-shot setting solely through metric evaluation of the language model. It has also demonstrated that, when faced with complex and unknown real-world environments, it is more reliable than supervised learning-based models.

Chapter 6

Conclusions

In conclusion, this dissertation has made significant contributions to the development and evaluation of assembly language models for security applications. By emphasizing the strengths of fine-tuning, showcasing the potential of zero-shot learning, and providing actionable insights, it lays a solid foundation for advancing instruction representation learning. These findings pave the way for more effective and reliable models in the domain of binary static analysis. We conclude our key contributions include the following:

To address unresolved challenges in instruction representation, we introduced PalmTree, a pre-trained assembly language model specifically designed to generate high-quality instruction embeddings. PalmTree employs self-supervised training on large-scale unlabeled binary corpora and incorporates newly designed tasks which are CWP (Context Window Prediction), and DUP (Def-Use Prediction). Experimental results demonstrate that PalmTree significantly outperforms existing models in intrinsic evaluations and down-

stream applications, such as function similarity search type inference, and value set analysis. These results underscore its utility in advancing binary analysis tasks.

Building on this foundation, we conducted a systematic evaluation of custom Transformer-based models and their associated pre-training tasks, including PalmTree, jTrans, StateFormer, and Trex, alongside the baseline BERT model. These models were assessed on four major downstream tasks: Function Similarity Search, Type Inference, Algorithm Classification, and Function Name Prediction. Our findings reveal that certain pre-training tasks, such as GSM, are excessively complex for effective learning, and architectural modifications alone do not provide substantial improvements. However, fine-tuning enhancements, like contrastive learning for Function Similarity Search, demonstrated more promising results, suggesting that focusing on fine-tuning strategies is often more impactful than introducing new pre-training tasks or altering model architectures.

Leveraging insights from PalmTree and other pre-trained models, we developed ALMOND, a zero-shot obfuscation detector powered by a transformer language model. ALMOND employs a metric-based classification technique augmented with innovative metrics like Error-Perplexity and Continuous Error-Prediction Penalty, achieving state-of-the-art performance. It demonstrated 96.3% accuracy on binaries with unseen obfuscation methods and excelled in real-world malware detection tasks, achieving a false negative rate of less than 0.001 and a false positive rate of just 0.269. These results highlight ALMOND's effectiveness in addressing obfuscation challenges, particularly in complex and previously unencountered scenarios.

6.1 Future Work

While this research has made substantial contributions to the field, it also highlights opportunities for future exploration. Adapting models to different assembly languages and architectures, applying transfer learning for cross-context performance, and collaborating with industry to refine real-world deployments are key directions. Additionally, leveraging large language model (LLM) techniques could significantly advance binary analysis. For example, instruction-tuned LLMs and in-context learning could enable more flexible workflows, while chain-of-thought prompting and causal reasoning could improve interpretability in tasks like vulnerability detection or control flow analysis. Integrating domain-specific knowledge and retrieval-augmented generation (RAG) techniques could further enhance model accuracy and applicability in areas such as malware detection and reverse engineering. By pursuing these directions, future research can unlock the full potential of LLMs in binary analysis.

Bibliography

[1] DeepVSA. `https://github.com/Henrygwb/deepvsa/`, 2019.

[2] EKLAVYA. `https://github.com/shensq04/EKLAVYA`, 2019.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[4] Christopher R Aberger. Recommender: An analysis of collaborative filtering techniques, 2016.

[5] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. ACM, 2013.

[6] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 361–374, 2022.

[7] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

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

[9] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[10] Fiorella Artuso, Marco Mormando, Giuseppe A Di Luna, and Leonardo Querzoni. Binbert: Binary code understanding with a fine-tunable and execution-aware transformer. *arXiv preprint arXiv:2208.06692*, 2022.

[11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. 2011.

[12] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Amir Bakarov. A survey of word embeddings evaluation methods. *CoRR*, abs/1801.09536, 2018.

[14] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19:31, 2005.

[15] Roberto Baldoni, Giuseppe Antonio Di Luna, Luca Massarelli, Fabio Petroni, and Leonardo Querzoni. Unsupervised features extraction for binary similarity using graph embedding neural networks. *arXiv preprint arXiv:1810.09683*, 2018.

[16] Imon Banerjee, Sriraman Madhavan, Roger Eric Goldman, and Daniel L Rubin. Intelligent word embeddings of free-text radiology reports. In *AMIA Annual Symposium Proceedings*, volume 2017, page 411. American Medical Informatics Association, 2017.

[17] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[18] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[19] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: a learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 3589–3601, 2018.

[20] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in neural information processing systems*, 31, 2018.

[21] Yoshua Bengio, Aaron C Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR, abs/1206.5538*, 1:2012, 2012.

[22] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[23] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[24] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. Effective, efficient, and robust packing detection and classification. *Computers & Security*, 85:436–451, 2019.

[25] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[26] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 2009.

[27] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35, 2016.

[28] Hongyun Cai, Vincent W Zheng, and Kevin Chang. A comprehensive survey of graph embedding: problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[29] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[30] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *AAAI*, pages 1145–1152, 2016.

[31] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689. ACM, 2016.

[32] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[33] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[34] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[35] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC, August 2017. USENIX Association.

[36] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *USENIX Security Symposium*, pages 99–116, 2017.

[37] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of bert models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE, 2021.

[38] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*, 2019.

[39] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[40] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2702–2711. JMLR.org, 2016.

[41] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[42] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

[43] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[44] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *ACM SIGPLAN Notices*, volume 52, pages 79–94. ACM, 2017.

[45] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *ACM SIGPLAN Notices*, volume 53, pages 392–404. ACM, 2018.

[46] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *Acm Sigplan Notices*, 49(6):349–360, 2014.

[47] Franck De Goër, Sanjay Rawat, Dennis Andriesse, Herbert Bos, and Roland Groz. Now you see me: Real-time dynamic function call detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 618–628. ACM, 2018.

[48] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[50] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[51] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys*, 27(3):326–327, 1995.

[52] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[53] Marius Drăgoi, Elena Burceanu, Emanuela Haller, Andrei Manolache, and Florin Brad. Anoshift: A distribution shift benchmark for unsupervised anomaly detection. *Neural Information Processing Systems NeurIPS, Datasets and Benchmarks Track*, 2022.

[54] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[55] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*, 2020.

[56] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[57] Thomas Dullein and Rolf Rolles. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de L'information et des communications*, 2005.

[58] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014.

[59] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[60] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 78–87. IEEE, 2014.

[61] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[62] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[63] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[64] F. Fouss, A. Pirotte, J. Renders, and M. Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, March 2007.

[65] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

[66] Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.

[67] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.

[68] Lian Gao, Yu Qu, Sheng Yu, Yue Duan, and Heng Yin. Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing. *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[69] Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

[70] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[71] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[72] Claudia Greco, Michele Ianni, Antonella Guzzo, and Giancarlo Fortino. Enabling obfuscation detection in binary software through explainable ai. *IEEE Transactions on Emerging Topics in Computing*, 2024.

[73] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[74] Yeming Gu, Hui Shu, and Fan Hu. Uniasm: Binary code similarity detection without fine-tuning. *arXiv preprint arXiv:2211.01144*, 2022.

[75] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 601–612. IEEE, 2022.

[76] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 601–612, 2022.

[77] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. {DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1787–1804, 2019.

[78] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: Facilitating value-set analysis with deep learning for postmortem program analysis. In *USENIX Security Symposium*, pages 1787–1804, 2019.

[79] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, Santa Clara, CA, August 2019. USENIX Association.

[80] Leonardo Gutiérrez-Gómez and Jean-Charles Delvenne. Unsupervised network embeddings with node identity awareness. *Applied Network Science*, 4(1):1–21, 2019.

[81] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[82] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[83] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022.

[84] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[85] Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. Binprov: Binary code provenance identification without disassembly. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 350–363, 2022.

[86] Mark Heimann, Haoming Shen, Tara Safavi, and Danai Koutra. Regal: Representation learning-based graph alignment. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 117–126. ACM, 2018.

[87] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162, 2018.

[88] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[89] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166. ACM, 2017.

[90] Ren-Hung Hwang, Min-Chun Peng, Chien-Wei Huang, Po-Ching Lin, and Van-Linh Nguyen. An unsupervised deep learning model for early network traffic anomaly detection. *IEEE Access*, 8:30387–30399, 2020.

[91] Félix Iglesias and Tanja Zseby. Analysis of network traffic features for anomaly detection. *Machine Learning*, 101:59–84, 2015.

[92] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*, pages 401–412, 2022.

[93] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2016.

[94] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[95] Shuai Jiang, Cai Fu, Yekui Qian, Shuai He, Jianqiang Lv, and Lansheng Han. Ifattn: Binary code similarity analysis based on interpretable features with attention. *Computers & Security*, 120:102804, 2022.

[96] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 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*, pages 1631–1645, 2022.

[97] Yu Jin and Joseph F. JáJá. Learning graph-level representations with gated recurrent neural networks. *CoRR*, abs/1805.07683, 2018.

[98] Ian Jolliffe. Principal component analysis. In *International encyclopedia of statistical science*, pages 1094–1096. Springer, 2011.

[99] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[100] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code artificiality: A metric for the code stealth based on an n-gram model. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 31–37, 2015.

[101] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE, 2020.

[102] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence: 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, December 4-6, 2004. Proceedings 17*, pages 488–499. Springer, 2005.

[103] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2022.

[104] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 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*, pages 151–163, 2022.

[105] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[106] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. *Advances in neural information processing systems*, 28:3294–3302, 2015.

[107] Sangjun Ko, Jusop Choi, and Hyoungshick Kim. Coat: Code obfuscation tool to evaluate the performance of code plagiarism detection tools. In *2017 International conference on software security and assurance (ICSSA)*, pages 32–37. IEEE, 2017.

[108] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. Semantic-aware binary code representation with bert. *arXiv preprint arXiv:2106.05478*, 2021.

[109] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[110] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

[111] Nathaniel Lageman, Eric D Kilmer, Robert J Walls, and Patrick D McDaniel. Bindnn: Resilient function matching using deep learning. In *International Conference on Security and Privacy in Communication Systems*, pages 517–537. Springer, 2016.

[112] Nathaniel Lageman, Eric D. Kilmer, Robert J. Walls, and Patrick Drew McDaniel. Bindnn: Resilient function matching using deep learning. In Robert Deng, Vinod Yegneswaran, Jian Weng, and Kui Ren, editors, *Security and Privacy in Communication Networks -12th International Conference, SecureComm 2016, Proceedings*, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST, pages 517–537, Germany, 1 2017. Springer Verlag.

[113] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[114] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[115] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[116] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[117] Van-Hoang Le and Hongyu Zhang. Log-based anomaly detection with deep learning: How far are we? In *Proceedings of the 44th international conference on software engineering*, pages 1356–1367, 2022.

[118] Aodong Li, Chen Qiu, Marius Kloft, Padhraic Smyth, Maja Rudolph, and Stephan Mandt. Zero-shot anomaly detection via batch normalization. *Neural Information Processing Systems NeurIPS*, 36, 2024.

[119] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

[120] Yao Li, Weiyang Xu, Yong Tang, Xianya Mi, and Baosheng Wang. Semhunt: Identifying vulnerability type with double validation in binary code. In *SEKE*, pages 491–494, 2017.

[121] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[122] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[123] Bang Liu, Ting Zhang, Di Niu, Jinghong Lin, Kunfeng Lai, and Yu Xu. Matching long text documents via graph convolutional networks. *arXiv preprint arXiv:1802.07459*, 2018.

[124] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αDiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, 2018.

[125] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678. ACM, 2018.

[126] Guangming Liu, Xin Zhou, Jianmin Pang, Feng Yue, Wenfu Liu, and Junchao Wang. Codeformer: A gnn-nested transformer model for binary code similarity detection. *Electronics*, 12(7):1722, 2023.

[127] Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. Logprompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 364–365, 2024.

[128] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[129] Farhana Ferdousi Liza and Marek Grześ. An improved crowdsourcing based evaluation technique for word embedding methods. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 55–61, 2016.

[130] Lajanugen Logeswaran and Honglak Lee. An efficient framework for learning sentence representations. *CoRR*, abs/1803.02893, 2018.

[131] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[132] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM, 2014.

[133] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[134] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[135] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533. CSREA Press, 2006.

[136] Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *International Conference on Learning Representations*, 2020.

[137] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 288–308. Springer, 2019.

[138] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.

[139] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[140] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, pages 1–11, 2019.

[141] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[142] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[143] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[144] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119. 2013.

[145] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[146] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

[147] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[148] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security Conference*, pages 416–430. Springer, 2015.

[149] O. Mirzaei, J.M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[150] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[151] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[152] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[153] Changan Niu, Chuanyi Li, Vincent Ng, David Lo, and Bin Luo. Fair: Flow type-aware pre-training of compiler intermediate representations. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[154] ORACLE. x86 Assembly Language Reference Manual. `https://docs.oracle.com/cd/E26502_01/html/E28388/ennbz.html`, 2019.

[155] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[156] Jonathan Pan, Wong Swee Liang, and Yuan Yidi. Raglog: Log anomaly detection using retrieval augmented generation. In *2024 IEEE World Forum on Public Safety Technology (WFPST)*, pages 169–174. IEEE, 2024.

[157] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[158] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering*, 49(04):2776–2790, apr 2023.

[159] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 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*, pages 690–702, 2021.

[160] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[161] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 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*, pages 747–759, 2022.

[162] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR, 2021.

[163] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[164] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM.

[165] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of NAACL-HLT*, pages 2227–2237, 2018.

[166] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[167] Raul Puri and Bryan Catanzaro. Zero-shot text classification with generative language models. *arXiv preprint arXiv:1912.10165*, 2019.

[168] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. volume 63, pages 1872–1897, 2020.

[169] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[170] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training (2018). *URL http://openai-assets.s3.amazonaws.com/research-covers/language-unsupervised/language_understanding_paper.pdf*, 2018.

[171] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[172] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[173] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware Detection by Eating a Whole EXE. In *AAAI-2018 Workshop on Artificial Intelligence for Cyber Security*, 2018.

[174] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959, 2009.

[175] Lukas Ruff, Jacob R Kauffmann, Robert A Vandermeulen, Grégoire Montavon, Wojciech Samek, Marius Kloft, Thomas G Dietterich, and Klaus-Robert Müller. A unifying review of deep and shallow anomaly detection. *Proceedings of the IEEE*, 109(5):756–795, 2021.

[176] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[177] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.

[178] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, New York, NY, USA, 2016. Association for Computing Machinery.

[179] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, New York, NY, USA, 2016. Association for Computing Machinery.

[180] Atsushi Sato and Keiji Yamada. Generalized learning vector quantization. *Advances in neural information processing systems*, 8, 1995.

[181] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Acm computing surveys (csur)*, 49(1):1–37, 2016.

[182] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[183] Eli Schwartz, Assaf Arbelle, Leonid Karlinsky, Sivan Harary, Florian Scheidegger, Sivan Doveh, and Raja Giryes. Maeday: Mae for few-and zero-shot anomaly-detection. *Computer Vision and Image Understanding*, 241:103958, 2024.

[184] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, 2016.

[185] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[186] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, Jessie Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[187] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. *Advances in neural information processing systems*, 29, 2016.

[188] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Mab-malware: A reinforcement learning framework for attacking static malware classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[189] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[190] Jianlin Su. Simbert: Integrating retrieval and generation into bert. Technical report, 2020.

[191] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. Pattern recognition techniques for the classification of malware packers. In Ron Steinfeld and Philip Hawkes, editors, *Information Security and Privacy*, pages 370–390, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[192] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27:3104–3112, 2014.

[193] Ke Tang, Zheng Shan, Fudong Liu, Yizhao Huang, Rongbo Sun, Meng Qiao, Chunyan Zhang, Jue Wang, and Hairen Gui. Srobr: Semantic representation of obfuscation-resilient binary code. *Wireless Communications and Mobile Computing*, 2022, 2022.

[194] Zhenzhou Tian, Hengchao Mao, Yaqian Huang, Jie Tian, and Jinrui Li. Fine-grained obfuscation scheme recognition on binary code. In Pavel Gladyshev, Sanjay Goel, Joshua James, George Markowsky, and Daryl Johnson, editors, *Digital Forensics and Cyber Crime*, pages 215–228, Cham, 2022. Springer International Publishing.

[195] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW9 '19, New York, NY, USA, 2019. Association for Computing Machinery.

[196] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673. IEEE, 2015.

[197] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[198] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[199] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[200] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2019.

[201] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234. ACM, 2016.

[202] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jtrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[203] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 319–330. IEEE Press, 2017.

[204] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[205] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th annual computer security applications conference*, pages 222–235, 2018.

[206] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[207] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 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*, pages 363–376, 2017.

[208] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 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*, pages 363–376. ACM, 2017.

[209] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*, pages 462–472. IEEE Press, 2017.

[210] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.

[211] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[212] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[213] Lee Young Jun, Choi Sang-Hoon, Kim Chulwoo, Lim Seung-Ho, and Park Ki-Woong. Learning binary code with deep learning to detect software weakness. In *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*, 2017.

[214] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, 41(3):793–819, 2014.

[215] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2709–2725, 2022.

[216] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[217] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33:3872–3883, 2020.

[218] Jingqing Zhang, Piyawat Lertvittayakumjorn, and Yike Guo. Integrating semantic knowledge to tackle zero-shot text classification. *arXiv preprint arXiv:1903.12626*, 2019.

[219] Yifan Zhang, Chen Huang, Yueke Zhang, Kevin Cao, Scott Thomas Andersen, Huajie Shao, Kevin Leach, and Yu Huang. Combo: Pre-training representations of binary code using contrastive learning. *arXiv preprint arXiv:2210.05102*, 2022.

[220] Yujie Zhao, Zhanyong Tang, Guixin Ye, Dongxu Peng, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. Semantics-aware obfuscation scheme prediction for binary. *Computers & Security*, 99:102072, 2020.

[221] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[222] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 2357–2374, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[223] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *26th Annual Network and Distributed System Security Symposium,*

NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society, 2019.

[224] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.