CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision

Hao Wang*
Tsinghua University
Beijing, China
hao-wang20@mails.tsinghua.edu.cn

Zeyu Gao*
Tsinghua University
Beijing, China
gaozy22@mails.tsinghua.edu.cn

Chao Zhang*†‡
Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

Zihan Sha
Information Engineering University
Zhengzhou, China
technicalgrit@foxmail.com

Mingyang Sun
University of Electronic Science and Technology of China
Chengdu, China
2020090918021@std.uestc.edu.cn

Yuchen Zhou
Beijing University of Technology
Beijing, China
zhouyuchen@emails.bjut.edu.cn

Wenyu Zhu
Tsinghua University
Beijing, China
zhuwy19@mails.tsinghua.edu.cn

Wenju Sun
Tsinghua University
Shenzhen, China
swj22@mails.tsinghua.edu.cn

Han Qiu*
Tsinghua University
Beijing, China
qiuhan@tsinghua.edu.cn

Xi Xiao
Tsinghua University
Shenzhen, China
xiaox@sz.tsinghua.edu.cn

ABSTRACT

Binary code representation learning has shown significant performance in binary analysis tasks. However, existing solutions often have poor transferability, particularly in few-shot and zero-shot scenarios where few or no training samples are available for the tasks. To address this problem, we present CLAP (**C**ontrastive **L**anguage-**A**ssembly **P**re-training) , which employs natural language supervision to learn better representations of binary code (i.e., assembly code) and get better transferability. At the core, our approach boosts superior transfer learning capabilities by effectively aligning binary code with their semantic explanations (in natural language), resulting in a model able to generate better embeddings for binary code. To enable this alignment training, we propose an efficient *data generator* that can automatically generate a large and diverse dataset comprising binary code and corresponding natural language explanations. We have generated **195 million** pairs of binary code and explanations and trained a prototype of CLAP. The evaluations of CLAP across various downstream tasks in binary analysis all demonstrate exceptional performance. Notably, without any task-specific training, CLAP is often competitive with a fully supervised baseline, showing excellent transferability.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Computing methodologies** → **Natural language processing**; • **Theory of computation** → **Program analysis**.

KEYWORDS

Deep Learning, Binary Analysis, Representation Learning

ACM Reference Format:
Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652145

1 INTRODUCTION

Deep learning is effective at numerous binary analysis tasks, including function boundary detection [51, 58, 73], binary code search [72], binary code similarity detection [15, 22, 36, 66, 77], function type inference [8], malware classification [55], reverse engineering [28, 61], binary code summarization [24], software composition analysis [21] and value set analysis [14]. The success of deep learning in the binary analysis field can be attributed to its powerful representation learning capabilities, which have been proven effective in capturing complex patterns and relationships within the data as well as learning meaningful representations of assembly code.

*Institute for Network Sciences and Cyberspace
†Zhongguancun Laboratory
‡Corresponding author: chaoz@tsinghua.edu.cn

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652145

Despite the progress made by deep learning techniques in binary analysis, existing methods suffer from certain limitations. First, current methods typically require substantial data for retraining when applied to new datasets or tasks, leading to poor performance in scenarios with sparse training samples. This is particularly problematic in few-shot learning scenarios, where a model must adapt to new tasks with minimal examples, and zero-shot learning scenarios, where the model encounters tasks it has never seen during training. Second, existing schemes for encoding assembly code often result in losing critical information, such as the parameters to call instructions, strings, and variable names. In particular, most approaches [10, 32, 52, 66] tend to normalize character text, constant numerical values, and external functions using special vocabulary tokens, which inadvertently leads to the omission of vital details.

Drawing inspiration from multi-modal learning, we observe that models such as CLIP [53] could learn better image representations by aligning visual modality concepts with human-comprehensible natural language. Similarly, we could conceptualize binary code as an analogous modality and explore the alignment between binary code and natural language, to develop more semantically profound representations of assembly code with better transferability. Specifically, we could use natural language (i.e., explanations of code semantics) as a supervision signal for learning representations of binary code (i.e., assembly code) by aligning them with pre-training. The resulting model is highly likely to obtain representations that encapsulate more semantic information about binary code. *To this end, we must address the following challenges*: (1) obtaining cost-effective aligned data for training and (2) preserving semantic information (e.g., control-flow and data-flow) extracted by existing binary code representation methods.

To address these challenges, we introduce a novel method named CLAP, which utilizes natural language supervision to learn transferable binary code representations. Specifically, regarding the first challenge, we present an efficient data generator that automatically generates a large and diverse dataset of assembly code and natural language explanations for model pre-training. Regarding the second challenge, we introduce a novel binary code representation learning backbone network that combines WordPiece tokenization (e.g. used by Llama [62]) and jump-aware embedding (e.g., used by jTrans [66]) designs. This network is capable of capturing both data flow and control flow information in binary code while preserving crucial information, such as function-call parameters, decompiler-processed strings, and variable names.

We have generated **195 million** pairs of aligned data and pre-trained a prototype model of CLAP, and evaluated it on several downstream tasks, including binary code similarity detection (BCSD), crypto-related functions identification, and protocol categorization. The results indicate that CLAP outperforms existing state-of-the-art (SOTA) solutions on these tasks, even without any further task-specific training or fine-tuning. In the BCSD task, when searching for functions within 10,000 functions, CLAP without fine-tuning achieves the highest ranking with an average Recall@1 rate of 83.3%. In comparison, the current leading supervised solution only achieves a rate of 57.1%. Regarding crypto-related function identification, CLAP performs slightly below the baseline in the zero-shot setting, but the fine-tuned version of CLAP largely surpasses the baseline by 17%. In the protocol categorization experiment, both the zero-shot and finetuned versions of CLAP surpass the best baseline by 6% and 23%, respectively. Furthermore, we conducted a case study to evaluate CLAP on real-world samples. The results demonstrate that CLAP has a remarkable application potential.

This paper principally explores a critical research question: *Can a model be trained in the field of binary analysis to effectively transfer acquired knowledge to various tasks, even when confronted with extremely limited or non-existent data?* We affirmatively respond with CLAP, our innovative methodology that bridges the gap between binary code and natural language representations. The principal contributions of our research are as follows:

(1) We introduce CLAP, which innovatively uses natural language as a supervisory signal to learn binary code representations. This method aligns assembly code with natural language and boosts transfer learning capabilities, especially in few-shot and zero-shot learning scenarios.
(2) We develop an efficient and scalable data generator capable of generating a comprehensive dataset of assembly code and corresponding natural language explanations. This dataset is instrumental in training models to represent assembly code with natural language supervision, bridging a significant gap in existing binary analysis methodologies.
(3) We conduct extensive experiments to demonstrate the effectiveness of our proposed CLAP method, which outperforms best baselines and shows remarkable transfer learning capabilities on various tasks. This showcases the model's versatility and effectiveness in binary analysis applications, setting a new benchmark in the field.
(4) We release our code[1] and CLAP model[2] for the research community to facilitate future research.

2 BACKGROUND AND RELATED WORKS

This section provides an overview of the key concepts and techniques relevant to CLAP, focusing on binary code representation and large language models.

2.1 Binary Code Representation

During the compilation process, high-level languages are transformed into assembly code (i.e. binary code), which is closer to machine hardware. Assembly code is more difficult to interpret compared to the source code, primarily because it lacks the clear abstractions present at the source code level, such as variable names and high-level logical structures. An example of assembly code is shown in Figure 12, which implements a bubble sort algorithm. Often in binary program analysis, there is a significant lack of access to the source code. This not only makes understanding the program more challenging but also spawns scenarios akin to the representation of the source code, underscoring the demand for effective representations of the assembly code.

Binary code representations are essential for various tasks such as binary code similarity detection (BCSD), function prototype inference, malware classification, and reverse engineering. Over the years, researchers have devised numerous techniques to represent binary code as continuous vectors in a vector space, making it

[1] https://github.com/Hustcw/CLAP
[2] https://huggingface.co/hustcw/clap-asm

suitable for downstream tasks. These methods for obtaining function vector representations can be broadly categorized into three groups: 1) direct modeling of raw bytes, 2) employing graph models to establish control flow relationships, and 3) representing function instructions as instruction sequences.

2.1.1 Raw Bytes. Several studies, such as MalConv [55], DeepVSA [14] and α-Diff [36], employ neural networks like CNNs [29] and LSTMs[17] to analyze binary code from raw bytes, aiming to improve malware detection and identify code similarities. These approaches focus on capturing data dependencies and feature extraction without delving into instruction-level semantics or control flow graph structures, offering efficient computational processing but missing out on deeper semantic understanding.

2.1.2 Graph Modeling. Assembly code, inherently structured as a series of basic blocks connected by Control Flow Graphs (CFGs), has prompted studies to model CFGs graphically. Gemini [70] utilizes GNNs to derive function embeddings from CFGs, though it is limited by the manual selection of features potentially missing complex semantics. GMN [33] innovatively calculates graph similarity via an attention-based Graph Matching Network. Both GraphEmb [9] and OrderMatters [64] infuse DNNs to learn basic block attributes before applying graph models to encapsulate the interrelations of these blocks within the CFG framework, capturing the flow and structure more effectively. VulHawk [43] integrates RoBERTa [39] and GCNs [26] for multi-faceted embeddings.

2.1.3 Sequence Modeling. Sequence modeling methods consider assembly code as an instruction series, capturing the instruction order. Asm2Vec [10] and SAFE [46] use language-inspired models to generate embeddings for instructions and functions, treating instructions analogously to words. Advanced techniques such as jTrans [66], Trex [52], BinShot [1], kTrans [77], CEBin [65] and BinaryAI [21] use Transformer to model assembly code. Nova$^+$ [22] treats the assembly code as normal text but continues pretrains and finetune the StarCoder [31] on a large corpus of binary data.

2.2 Large Language Models (LLM)

Recently, large language models especially the GPT series [3, 4, 54] of models, such as GPT-3.5 (i.e. ChatGPT) [49] and GPT-4 [48], have demonstrated extraordinary source code understanding capabilities and being applied to various software engineering problems [6, 11, 13, 19, 23, 41, 56, 60]. For instance, we prompt ChatGPT to output the explanation of the code snippet shown in Figure 1a, ChatGPT demonstrates its robust comprehension of source code not only through practical usage but also in its ability to elucidate code functions, i.e. "The purpose of this code is to create a string that represents the mode of a file or directory like -rw-r–r–".

In contrast, these models exhibit less proficiency with assembly code. When tasked to explain the assembly code for `xstrmode` shown in Figure 1b, ChatGPT is largely ineffective. This can be attributed to the challenges in capturing control flow information and the scarcity of assembly code in pre-training corpora.

2.3 Multi-Modality Learning

Multi-modality learning, merging computer vision (CV) and natural language processing (NLP), has made significant strides in recent

```
char *xstrmode(mode_t mode, char *str) {
    unsigned short i = 0;

    if (S_ISDIR(mode)) str[i++] = 'd';
    else if (S_ISLNK(mode)) str[i++] = 'l';
    else if (S_ISCHR(mode)) str[i++] = 'c';
    else if (S_ISBLK(mode)) str[i++] = 'b';
    else if (S_ISSOCK(mode)) str[i++] = 's';
    else if (S_ISFIFO(mode)) str[i++] = 'p';
    else if (S_ISREG(mode)) str[i++] = '-';

    str[i++] = mode & S_IRUSR ? 'r' : '-';
    str[i++] = mode & S_IWUSR ? 'w' : '-';
    str[i++] = (mode & S_ISUID
                ? (mode & S_IXUSR ? 's' : 'S')
                : (mode & S_IXUSR ? 'x' : '-'));
    str[i++] = mode & S_IRGRP ? 'r' : '-';
    str[i++] = mode & S_IWGRP ? 'w' : '-';
    str[i++] = (mode & S_ISGID
                ? (mode & S_IXGRP ? 's' : 'S')
                : (mode & S_IXGRP ? 'x' : '-'));
    str[i++] = mode & S_IROTH ? 'r' : '-';
    str[i++] = mode & S_IWOTH ? 'w' : '-';
    str[i++] = (mode & S_ISVTX
                ? (mode & S_IXOTH ? 't' : 'T')
                : (mode & S_IXOTH ? 'x' : '-'));
    str[i] = '\0';

    return str;
}
```

(a)

```
    mov rax, rsi
    mov esi, edi
    mov ecx, edi
    and esi, 61440
    cmp esi, 16384
    jne .L2
    mov BYTE PTR [rax], 100
    jmp .L3
.L2:
    cmp esi, 40960
    jne .L4
    mov BYTE PTR [rax], 108

// ... more assembly here ...

.L20:
    cmp esi, 1
    sbb ecx, ecx
    and ecx, -75
    add ecx, 120
.L21:
    add edx, 9
    movzx esi, di
    movzx edx, dx
    mov BYTE PTR [rax+rsi], cl
    mov BYTE PTR [rax+rdx], 0
    ret
```

(b)

Figure 1: A real-world function named `xstrmode` (a) and its assembly code (b)

years. Models like CLIP [53] have revolutionized this space by using text-image pairs for contrastive learning, enabling impressive zero-shot capabilities and aligning text with images. Other models, such as Flamingo [2], BLIP [30] and LLaVA [37], integrate CV and NLP through cross-attention or projection mechanisms, facilitating complex tasks like image-based dialogues. In code understanding, approaches like CodeT5+ [68, 69] have successfully applied similar multi-modal concepts, unifying text and source code processing. These advancements show the potential of multi-modal learning to bridge the gap among modalities.

3 METHODOLOGY

Figure 2 presents the core workflow of our study, detailing the essential stages of our process. Our methodology begins with extensive compilation activities on a 96-core server using the Ubuntu repository [5], which lasted three months. This process produced a diverse range of assembly codes, generated via various optimization techniques and compilers while retaining their corresponding source codes. For initial interpretations at the source code level, we utilized GPT-3.5, employing tailored prompts to bootstrap our data. Subsequently, we fine-tuned the LLaMA model [62] with this curated dataset. This fine-tuning was aimed at customizing the model's output to our specific requirements and performing local inference, which resulted in an expanded set of source code explanations for data augmentation.

To overcome the challenges outlined in Section 1, we introduce an innovative assembly encoder model, meticulously designed to capture both the semantic and structural nuances of assembly code. Initially, we pre-trained a jump-aware transformer on an extensive assembly code dataset, aiming to create an expressive and effective encoder model. Then, employing a contrastive learning strategy with a multitude of negative samples, we effectively aligned this assembly code encoder with a text encoder, which itself was pre-trained on more than 1 billion text pairs [57]. Through the optimization of the InfoNCE loss, we significantly enhanced the

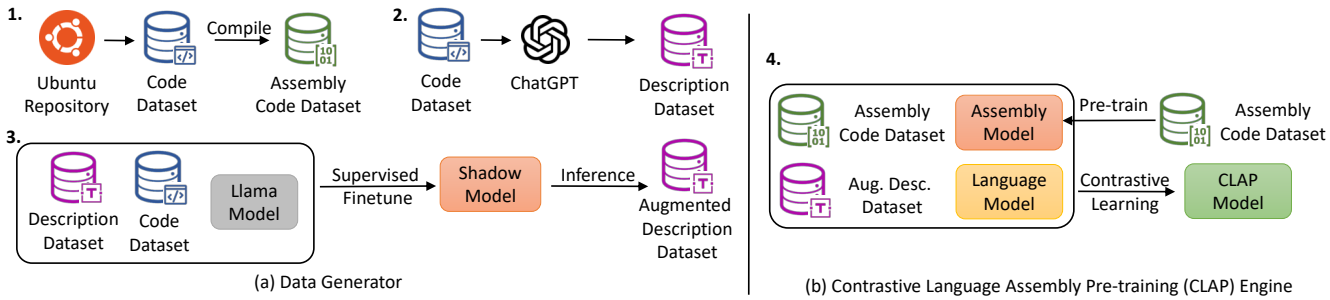


Figure 2: Overview of our primary workflow, consisting of two main components, the data generator, and the CLAP engine. The data generator compiles source code from the Ubuntu Repository into assembly code and uses GPT to generate explanations, forming an explanation dataset. We then fine-tune an LLaMA model using the source code with the corresponding explanation, resulting in a shadow model that generates the augmented explanation dataset. Within the CLAP engine, we first pre-train the assembly code dataset to develop an initial assembly encoder. By employing contrastive learning with this assembly encoder and a text encoder on both assembly code and augmented explanation dataset, we produce the final CLAP model.

mutual information between the positive and negative samples of assembly code and explanations. This optimization facilitated the precise matching of assembly code with text explanations and the successful alignment of their embeddings.

3.1 Data Generator

In this section, we will discuss the process of obtaining the training dataset. The core idea is to utilize LLMs' understanding of source code and construct a relationship between source code and assembly code to obtain natural language explanations for assembly code. Considering the cost of building the dataset, apart from using GPT, we locally trained a model capable of generating sufficiently good natural language explanations to create a large and diverse dataset.

3.1.1 Binary Generation. We utilize package managers to achieve large-scale automated compilation and source code acquisition, specifically using Ubuntu's package manager. Ubuntu is one of the most popular distributions, containing many C/C++ packages, each with a script that compiles the source code into the final product. This makes it suitable for generating a diverse set of binary executable files along with their source code.

We employ a variety of compilers and optimization levels to augment the quantity of assembly code, which contains six different compilers (`GCC-{7,9,11}` and `Clang-{9,11,12}`) and five optimization levels (`O{0-3}` and `Os`). Additionally, we maintain both stripped and non-stripped functions to facilitate assembly pre-training and contrastive pre-training in later stages.

We utilize Clang [40] to parse the preprocessed source code, extracting source code snippets by determining the position of each function within the code file. To enhance the semantic value of the functions and minimize the noise in the dataset, we exclude source code segments with less than three lines and eliminate assembly code containing less than three basic blocks.

3.1.2 Source Code Explanation. We employ GPT-3.5 to generate natural language explanations for source code, as it lacks comprehension of assembly code. By leveraging the correspondence between source code and assembly code, we aim to create sufficiently detailed text explanations that can explain the source code and assembly code at the same time.

We manually design the instructions for GPT-3.5 to generate an explanation. We first prompt the model to concisely summarize the source code while avoiding detailed procedural narrations. Secondly, acknowledging the intrinsic information loss during code compilation, such as the absence of variable names, we instruct the model to forgo these non-recoverable specifics. Lastly, we program the model to assign tags to the code snippet.

3.1.3 Shadow Model. We conduct supervised finetuning (SFT) on the Vicuña 13B [7] using half of the 2.6 million function explanations from GPT-3.5. For comparison, we train a Vicuña 13B and an LLaMA 30B model respectively, using 50,000 explanations of source code function. We evaluate and compare their performances in Section 4.4. After obtaining the shadow model, we employ it to facilitate the generation of a larger corpus of natural language explanations. This method serves as a data augmentation technique, enabling us to produce an expanded dataset of natural language explanations and assembly code pairs. As a result, the enriched dataset significantly increases the quantity of available data for Section 3.2.3 while maintaining acceptable quality.

3.2 Contrastive Language Assembly Pre-training (CLAP) Engine

Next, we will introduce the methodology of the CLAP engine, including the design of the assembly encoder (i.e., `CLAP-ASM`), the training process of the CLAP engine, and the rationale behind using natural language as a supervisory signal to enhance transferability.

3.2.1 Model Architecture. `CLAP-ASM` is based on the RoBERTa base architecture [39] with 110M parameters. We average the output of the last Transformer layer as the assembly embedding. As assembly code is quite different from natural languages to 1) effectively represent the assembly code, 2) capture its inherent structure and semantics and 3) preserve important information such as function-call parameters, decompiler-processed strings, and variable names,

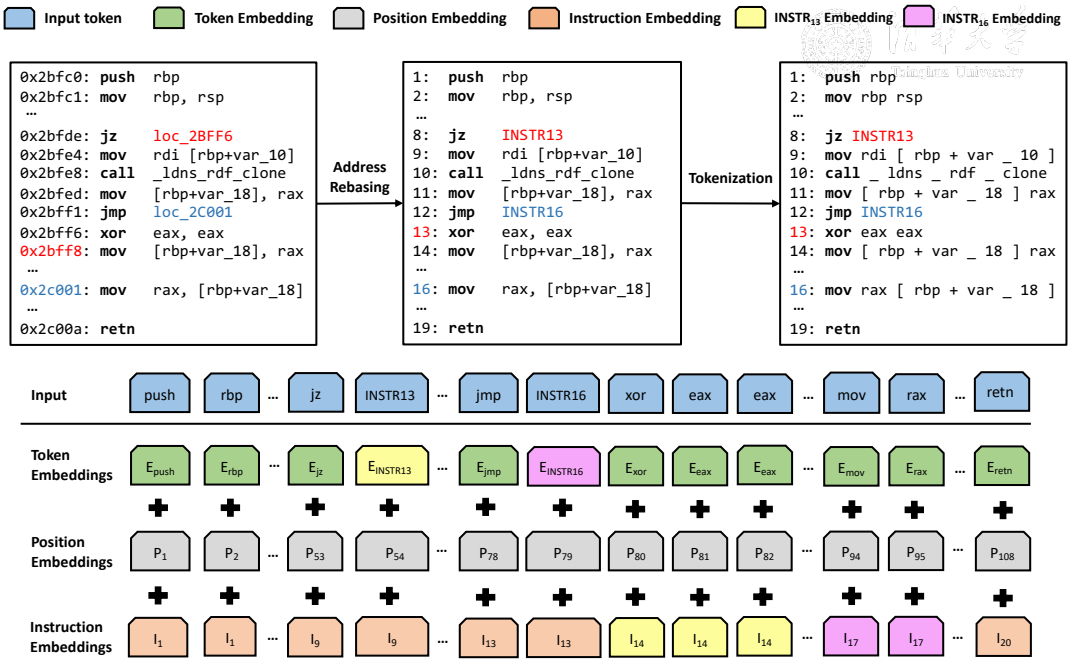


Figure 3: Illustration of CLAP-ASM. The raw assembly code is first rebased and tokenized. Then, each token is converted to a *token embedding*, a *position embedding*, and an *instruction embedding*, and the sum of these three is output as the embedding. The instruction embedding remembers instruction boundaries and works with jump symbols to comprehend control flow. The token embedding of the jump symbol (e.g., $\mathrm{E}_{\texttt{INSTR13}}$) shares parameters with the target instruction embedding (e.g., I_{13}).

we have adopted the following strategies in designing the assembly encoder. The CLAP-ASM is illustrated in Figure 3.

Instruction Embedding. We incorporate instruction embedding into the Transformer model to delineate the boundaries of assembly instructions. This allows the model to identify and process individual instructions effectively. By assigning unique embeddings to each instruction, the model learns to differentiate between various instructions. This distinction is crucial for the next procedures.

Address Rebasing. As shown in Figure 3, we first rebase the address of the assembly code, and we preserve the relative address relationships while handling jump instructions. Instead of normalizing different addresses to the same token like in previous work, this approach retains the control flow information within the assembly code. Address rebasing keeps the relative distance between jump instructions and their targets, enabling the model to understand the control flow changes caused by jump instructions.

Tokenization. After address rebasing, we tokenized the assembly code to separate tokens. We first employ the WordPiece [27] algorithm to train a tokenizer on the whole assembly code datasets, which is specifically tailored for assembly code. This tokenizer can perform a lossless encoding of assembly code without normalization, thus preserving crucial information such as calling parameters and external function names. In this way, we enable the model to handle the diverse vocabulary of assembly languages, including opcodes, registers, constants, and literals.

Jump Relationships. We enhance the Transformer's understanding of jump relationships within the assembly code by sharing the token embedding parameters of jump symbols and their corresponding instruction embeddings as shown in Figure 3. For example, the token embedding $\mathbf{E}_{\texttt{INSTR16}}$ shares parameters with the instruction embedding I_{16}. This shared representation aids the model in capturing the complex connections between various components of the assembly code. By sharing the parameters, the model learns to associate symbol tokens with their respective target tokens, allowing it to better understand the control flow within the code.

3.2.2 *Assembly Encoder Pre-training.*

We first pre-train the assembly encoder to comprehend assembly code by selectively masking tokens within the assembly context, which has been proven highly effective in previous work. We follow the same training task of previous work jTrans [66], including the Masked Language Model (MLM) task and the Jump Target Prediction (JTP) task. The collective loss function $\mathcal{L}_P(\theta)$ in the pre-training phase is the combination of MLM and JTP objective functions in Equation 1, where x_i is the i-th token, $\mathbf{m_x}$ and $\mathbf{l_x}$ denotes the masking positions for normal tokens and jump tokens.

$$\min_{\theta} \mathcal{L}_P(\theta) = \sum_{i \in \mathbf{m_x}} -\log P(x_i|\mathbf{f}^{\text{mlm}}) + \sum_{i \in \mathbf{l_x}} -\log P(x_i|\mathbf{f}^{\text{jtp}}) \quad (1)$$

Through the first stage of pre-training, we obtain a well-initialized set of weights for the second stage of contrastive pre-training, facilitating a hastened convergence of the contrastive pre-training.

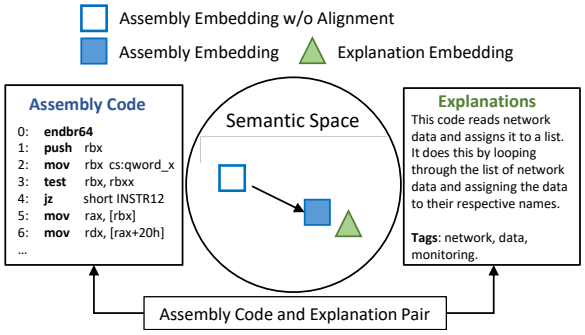


Figure 4: Given the assembly code and its corresponding textual explanation, we train the model to make the assembly code embedding and explanation embedding close through contrastive learning.

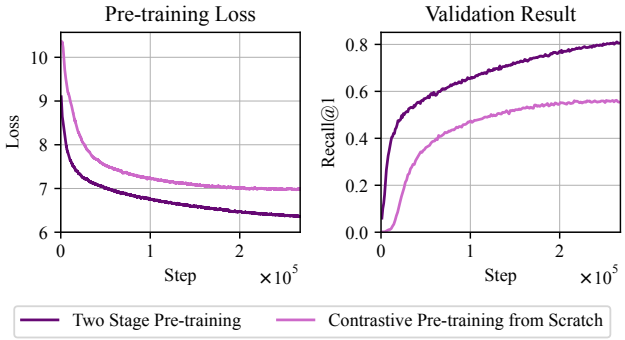


Figure 5: The comparison between contrastive pre-training from scratch and contrastive pre-training based on the first pre-training stage model (batch size = 1024, epoch <= 1). The left figure shows the InfoNCE loss during pre-training. The right figure shows the Recall@1 result in the validation dataset, in which the model needs to select the sole natural language explanation that matches the assembly code from 65,536 natural language explanations.

3.2.3 Contrastive Pre-training with Natural Language Supervision. After pre-training, we utilize a contrastive learning approach to align the assembly code encoder with a text encoder pre-trained on a large collection of text pairs [57]. The key to implementing natural language supervision with contrastive learning lies in using the natural language representations obtained from pre-trained natural language models as anchors. The training task is to predict which explanation corresponds to which assembly code. We present an example in the semantic space in Figure 4, demonstrating our training objective.

We apply the InfoNCE loss [47] to discriminate assembly code with positive and negative text explanations. Given a positive pair assembly code and text explanation (a_i, t_i) and a set of negative pairs $(a_i, t_j)_{j \neq i}$. We obtain representations for assembly code and explanations through assembly encoder $\mathcal{F}_A$ and text encoder $\mathcal{F}_T$, respectively. Through contrastive learning, we maximize the similarity between positive sample pairs while minimizing the similarity between the negative pairs.

$$E_{A_i} = \mathcal{F}_A(a_i), \quad E_{T_i} = \mathcal{F}_T(t_i) \tag{2}$$

where the assembly embedding for assembly code A_i denotes E_{A_i}, text embedding for explanation T_j denotes E_{T_i} and N denotes the number of samples in a single batch. The InfoNCE loss is defined as Equation 3

$$\mathcal{L}_E = -\log \frac{\exp(E_{A_i} \cdot E_{T_i})}{\sum_{j=1}^{N} \exp(E_{A_i} \cdot E_{T_j})} \tag{3}$$

By optimizing the InfoNCE loss, we enhance the mutual information between assembly code and explanations, effectively aligning their representations. In the implementation, we employ a well-pre-trained text encoder $\mathcal{F}_T$ and freeze its parameters, allowing for an exceptionally large batch size of $N = 65,536$. This approach enables our model to accurately identify the most fitting explanation from a vast pool of options for each assembly code. It allows our model to leverage the rich semantics of natural language explanations to augment the learned assembly code representations.

We compare contrastive learning from scratch and contrastive learning based on the first-stage model. The contrastive pre-training loss and the validation result are illustrated in Figure 5. The loss of the latter model decreases more rapidly during training and it achieves higher validation results. Compared to contrastive pre-training, the first pre-training stage incurs less than 10% of the cost, highlighting the efficiency of the two-stage pre-training.

3.3 Zero-Shot Inference Capability

As we introduce the rich semantic content of natural language into the representation of assembly code through contrastive learning, we can achieve impressive zero-shot capabilities in downstream assembly code understanding tasks. We demonstrate in Figure 6 how we use zero-shot for downstream tasks of assembly code understanding. In a multi-class classification task, the assembly code is first encoded into an embedding A_1 using `CLAP-ASM`. Concurrently, n prompts are constructed to describe the multi-class, which are subsequently processed by `CLAP-Text` to generate n corresponding text embeddings T_i. To derive zero-shot results, we calculate the dot product of A_1 and each T_i to compute the similarity. The prompt with the highest similarity is marked as the classification result. With each similarity as logits, we can apply the SoftMax function on $[A_1 \cdot T_1, A_1 \cdot T_2, \cdots, A_1 \cdot T_n]$ to get the probability of each label.

4 EVALUATION

We implement `CLAP` using Pytorch 2.0 [50]. We use IDA Pro 7.6 [16] to disassemble and extract the functions from the binary code in all of the experiments. Our training and experiments are conducted on several servers to accelerate training. The CPU setup is 128 cores with 2TB RAM for each server. The total GPU setup is 32 NVIDIA Tesla A100. We conduct extensive experiments to address the following questions:

- **RQ1:** How is the quality of the assembly code representations learned by `CLAP`?
- **RQ2:** Does the representation learned by `CLAP` exhibit strong transferability, even when confronted with extremely limited or non-existent data?

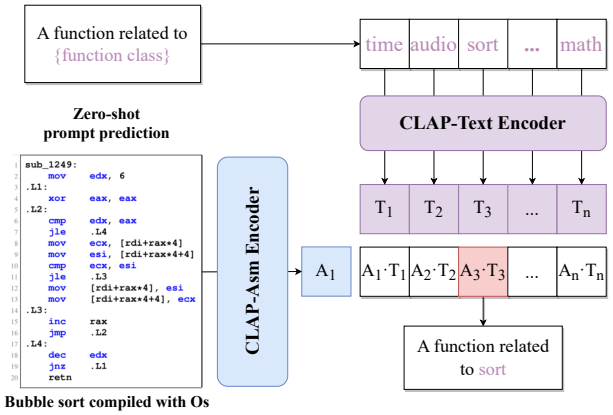


Figure 6: Zero-Shot Inference of CLAP

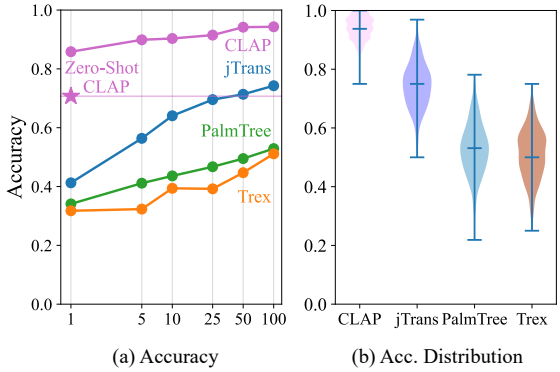


Figure 7: (a) Accuracy of jTrans, PalmTree, and CLAP in crypto identification task with different percentages of the training set. The x-axis shows the ratios of the training set (%). (b) Accuracy distribution with full training set finetuning.

- **RQ3:** Why natural language explanations could support binary code representation learning?
- **RQ4:** What is the quality of the dataset obtained by the data generator?

4.1 Quality of CLAP Representations (RQ1)

In representation learning, the efficacy of different models' representation is typically evaluated by freezing the pre-trained model and training only a linear projection layer for downstream task assessment, known as linear probe [38]. In our evaluation, we scrutinize a range of downstream tasks, notably the well-studied BCSD task [9, 52, 64, 66], along with various classification tasks. This methodology aims to confirm that the CLAP Model has effectively acquired a more sophisticated representation of assembly code.

4.1.1 Binary Code Similarity Detection (BCSD). BCSD represents a crucial approach to software security. It attempts to identify the degree of similarity between two assembly codes. This method is integral to binary analysis, with significant research dedicated to augmenting its accuracy and efficiency, essential for tasks such as vulnerability identification, malware detection, and supply chain analysis. According to [44], deep learning-based methods [10, 34, 46, 75] now have been becoming the de facto SOTA for BCSD, outperforming the traditional methods. We evaluate CLAP and several baselines using the `BinaryCorp` [66] datasets. The baselines include Gemini [71], GNN [34], GraphEmb [45], OrderMatters [74], SAFE [46], Asm2Vec [10], Trex [52], PalmTree [32] and jTrans [66].

To assess the effectiveness of the embeddings generated by pre-trained models, we incorporate a Linear Probe [53] into their respective outputs, enabling the linear layer to project these outputs into an appropriate vector space. More specifically, we add a linear classifier (without non-linear activation) to the top layer of the model and finetune it exclusively for particular tasks to evaluate the quality of the embeddings produced by the pre-trained models. Additionally, we provide the full finetune result of jTrans as it performs best during linear probing. To showcase the transferability of our model, we do not finetune CLAP. Instead, we utilize the model without any further training for comparison with the baselines.

The results of our experiments are presented in Table 1. We use Recall@1 and MRR (mean reciprocal rank) as metrics, which are used in previous work [66]. All baselines are evaluated with a pool size of 10,000, which means there is only one positive sample among 10,000 functions, which are sampled randomly from the whole dataset. Our results indicate that CLAP outperforms the closest baseline competitor by 0.244 for the MRR metric and by over 26.5% for the recall@1 metric on average across different tasks, which is even a finetuned model.

These results demonstrate that even in a zero-shot format, CLAP can significantly outperform state-of-the-art approaches like jTrans and Trex. We also evaluate jTrans in a zero-shot format, where it is pre-trained with the MLM and JTP approach, and the result is shown as jTrans (Zero Shot) in Table 1. The results show that CLAP outperforms jTrans in the zero-shot format by 0.453 for the MRR metric and by over 49.6% for the recall@1 metric. This reveals that contrastive learning with natural language supervision yields better assembly code representations compared to assembly language models that employ only self-supervised learning.

4.1.2 Crypto Identification. The crypto identification task is used to identify which cryptographic algorithm the assembly code belongs to. In this experiment, we first filter out the packages that are recognized as encryption algorithms from the Ubuntu repository and extract the functions. For each function, we classify it as an encryption algorithm based on its name and the meta information contained in the corresponding package. It is worth noting that, we ensure the functions are not presented in the training set of constrastive learning to avoid information leakage. We filter out functions with the number of basic blocks smaller than 3, which means that these functions possibly do not contain useful information. Finally, we obtain the dataset containing 17 types of encryption algorithms and about 70K functions. For comparison, jTrans [66], Trex [52], and PalmTree [32] are used as pre-trained baselines.

To adapt to the crypto identification downstream tasks, we evaluate the models using the linear probe method. We finetune the models with 1%, 5%, 10%, 25%, 50%, and 100% of training data, respectively. Figure 7 shows the performance of CLAP and each baseline We can see that CLAP achieves a high accuracy of 0.94 on the dataset with 50% training set and all training data. And even

Table 1: Comparison between CLAP and baselines for the BCSD task on BinaryCorp-3M dataset (Poolsize=10,000)

	MRR							Recall@1						
Models	**O0,O3**	**O1,O3**	**O2,O3**	**O0,Os**	**O1,Os**	**O2,Os**	**Average**	**O0,O3**	**O1,O3**	**O2,O3**	**O0,Os**	**O1,Os**	**O2,Os**	**Average**
Gemini	0.037	0.161	0.416	0.049	0.133	0.195	0.165	0.024	0.122	0.367	0.030	0.099	0.151	0.132
GNN	0.048	0.197	0.643	0.061	0.187	0.214	0.225	0.036	0.155	0.592	0.041	0.146	0.175	0.191
OrderMatters	0.062	0.319	0.600	0.075	0.260	0.233	0.263	0.040	0.248	0.535	0.040	0.178	0.158	0.200
GraphEmb	0.087	0.217	0.486	0.110	0.195	0.222	0.219	0.050	0.154	0.447	0.063	0.135	0.166	0.169
SAFE	0.127	0.345	0.643	0.147	0.321	0.377	0.320	0.068	0.247	0.575	0.079	0.221	0.283	0.246
Asm2Vec	0.072	0.449	0.669	0.083	0.409	0.510	0.366	0.046	0.367	0.589	0.052	0.332	0.426	0.302
PalmTree	0.130	0.403	0.677	0.152	0.355	0.496	0.369	0.08	0.326	0.609	0.097	0.281	0.420	0.302
Trex	0.118	0.477	0.731	0.148	0.511	0.513	0.416	0.073	0.388	0.665	0.088	0.422	0.436	0.345
jTrans (Zero Shot)	0.137	0.490	0.693	0.182	0.472	0.510	0.414	0.088	0.412	0.622	0.122	0.393	0.430	0.340
jTrans (Linear Probe)	0.333	0.573	0.715	0.404	0.608	0.601	0.539	0.245	0.494	0.644	0.309	0.526	0.520	0.456
jTrans (Finetune)	0.475	0.663	0.731	0.539	0.665	0.664	0.623	0.376	0.580	0.661	0.443	0.586	0.585	0.571
CLAP (Zero shot)	**0.764**	**0.903**	**0.941**	**0.813**	**0.906**	**0.877**	**0.867**	**0.719**	**0.875**	**0.920**	**0.774**	**0.881**	**0.847**	**0.836**

when trained on only 1% of the data, CLAP's performance surpasses the results of the other two models trained on all training data. This strongly proves our model's excellent generalization ability and effectiveness. We also point out the zero-shot performance (More explanation in Section 3.3 and 4.2) in Figure 7a.

We also record the accuracy of each batch during the evaluation process by the model that trained on all training data and explore the data distribution characteristics of experimental results by drawing violin plots. From Figure 7b, it can be seen that for CLAP, the accuracy is mainly concentrated in a higher range of values. This indicates that our model can accurately classify samples in most cases. Compared to the baselines, CLAP has better stability and more concentrated results.

4.1.3 Protocol Categorization. Similar to the preprocessing and experiment in Section 4.1.2, we use the packages marked as protocol and obtain a dataset of 736K functions, containing 18 protocol types.

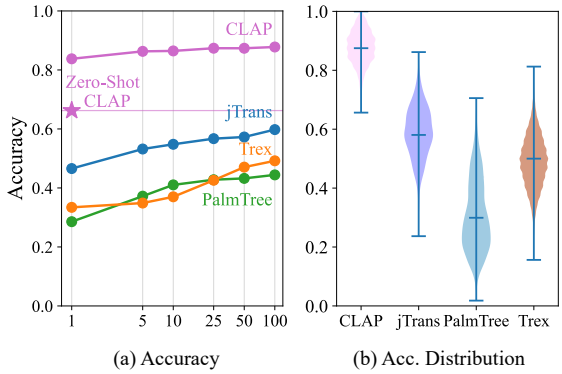
Figure 8a presents the results in terms of accuracy. It indicates that our method, in the crypto identification task, finetuned on a mere 1% of data can surpass the performance of other baselines finetuned on the full dataset. Furthermore, even the zero-shot approach alone outperforms or approaches other models finetuned on the full dataset, highlighting our model's superior capability in obtaining an outstanding assembly code representation. The violin plot of accuracy distribution is shown in Figure 8b, showing an impressive and stable distribution similar to the one in Section 4.1.2.

The conclusion can be drawn by combining the analysis of the aforementioned aspects, indicating that our model acquires a superior representation of assembly code and demonstrates greater stability in its learning results.

4.2 Transferability Evaluation (RQ2)

Zero-shot learning allows the model to make predictions or solve novel tasks without having seen any examples from the task's specific class beforehand, while few-shot learning enables the model to adapt quickly to new tasks with only a small number of examples. In this evaluation, we strive to assess the model's zero-shot and few-shot capabilities to validate its transferability performance.

4.2.1 Zero Shot. A major benefit of aligning assembly code with natural language explanation is the capacity to seamlessly interact with assembly code using natural language. We explore the model's zero-shot capabilities to highlight this benefit. We use the method introduced in Section 3.3 to conduct the zero-shot experiments.



(a) Accuracy (b) Acc. Distribution

Figure 8: (a) Accuracy of CLAP, jTrans, PalmTree, and Trex in protocol categorization task with different percentages of the training set. The x-axis shows the ratios of the training set (%). (b) Accuracy distribution with full training set finetuning.

4.2.2 Few Shot. We acknowledge the challenges in obtaining data for binary tasks, as they often necessitate manual analysis to acquire a limited number of samples. To simulate this, we provide each model with a small set of samples and employ the linear probe technique. To compare the few-shot capability of each baseline, we train the model using the linear probe method with a small dataset, in which each label has 1, 2, 4, 8, or 16 samples. Due to the limited number of samples, the selection of the samples may have a significant impact on the final results, so we repeat the experiments for each sample size five times.

4.2.3 Results. The results in the Figure 9 indicate exciting results. When using only a single sample for training each label, accuracy often falls short compared to zero-shot learning, potentially due to overfitting. This demonstrates that our model can achieve satisfactory results even without samples. It is only when there is an ample number of samples for every category that few-shot learning results surpass those of zero-shot learning. We attribute this difference to the way each method interacts with assembly code features: while zero-shot learning allows natural language to directly engage with

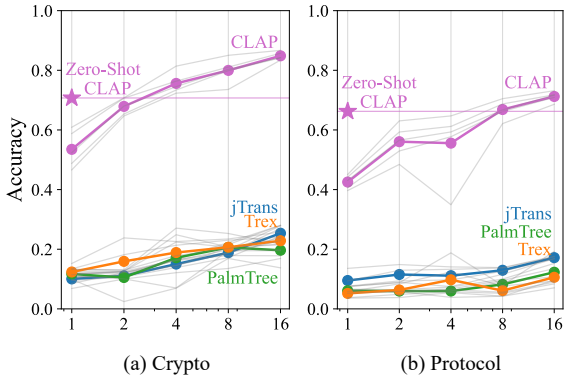


Figure 9: Accuracy of different few samples on Crypto Identification (a) and Protocol Categorization (b). The x-axis shows the number of labeled training examples per class. Each few-show experiment is conducted 5 times. The grey lines show the accuracy of an individual trial. The colored lines show the average accuracy across different trials.

these features, conventional few-shot learning relies on indirect interaction, resulting in difficulty when learning with limited samples. In contrast, the other baselines are even struggling to learn anything. The accuracy for the baselines is all below 0.3.

Therefore, the above results significantly demonstrate that CLAP has excellent zero-shot and few-shot capability, thus exhibiting remarkable transferability even in scenarios where data is severely constrained or absent.

4.3 Explaining Natural Language's Role in Assembly Representation Learning (RQ3)

To investigate the role of natural language supervision in assembly code representation, we conduct some intrinsic evaluations to study how the assembly code representation changes before and after alignment with natural language representations.

4.3.1 Representation Visualization. We first utilize t-SNE [63] to visualize the assembly representation and enhance our understanding of how the CLAP model effectively learns from natural language supervision. As a tool renowned for transforming high-dimensional data into more comprehensible two or three-dimensional formats, t-SNE provides a clear and intuitive view of data relationships, thereby illustrating the nuanced alignment achieved by our model.

For the experiment, we manually select ten functionality labels, including `time`, `encryption`, `authentication`, `search`, `math`, file `management`, `graphics`, `networking`, `sorting`, and `audio`. From real-world programs, we carefully handpick 50 functions that exemplify each functionality, resulting in a dataset called the "Intrinsic Dataset" comprising 500 functions. Next, the functions are encoded by five models: Trex, PalmTree, jTrans, and the CLAP-ASM model both with and without alignment. The t-SNE algorithm then reduces these representations to two dimensions, facilitating visualization shown in Figure 10.

In the visualization, we observe that the embeddings generated by the CLAP-ASM without alignment, as well as those generated by Trex, PalmTree, and jTrans all display a nearly disordered distribution. This pattern of disarray is consistent across these models, indicating a lack of high-level program semantics in their embeddings. In contrast, after natural language supervision which aligns with text embedding, the CLAP-ASM model's output can be reduced to the same vector space according to the functionality, displaying similar distribution characteristics. This visual evidence underscores the alignment effect between our model and the natural language model, implying the potential for manipulating assembly language models using natural language directly in a zero-shot scenario.

4.3.2 Insights of Natural Language Supervision. After the implementation of contrastive learning enhanced by natural language supervision, we observe a notable performance improvement. This enhancement can be attributed to the introduction of essentially unlimited labels via sophisticatedly trained text encoders. Unlike contrastive learning on the solitary modality of assembly code, the incorporation of natural language signals offers pivotal anchors throughout the training process. This approach enriches the training context and cultivates features with superior transferability.

To elucidate our findings, consider a scenario with two cryptographic algorithms' assembly code implementations: RSA, an asymmetric encryption algorithm, and MD5, a hash function. Contrastive learning, solely reliant on assembly code, typically enables the model to identify similarities within RSA and MD5 implementations independently, but it struggles to link the two.

Introducing natural language as a supervisory signal changes this dynamic. For example, the natural language descriptions "RSA, a modern asymmetric encryption algorithm in cryptography" and "MD5, a widely used cryptographic hash function" serve as multifaceted labels. These labels associate RSA with the terms 'cryptography, asymmetric encryption, RSA' and MD5 with 'cryptography, hash algorithm, MD5', thereby creating a connective thread between the two under the broad theme of cryptography. This highlights the pivotal role of natural language in establishing meaningful connections in assembly code representations.

4.4 Data Generator Evaluation (RQ4)

In this section, we evaluate the quality of the explanation generated by GPT and Shadow Model, i.e., whether and how well the explanation matches the source code.

4.4.1 Quality of Source Explanation. To assess the quality of source code explanations, we engage five domain experts to independently review and score the explanations from GPT-3.5 and the shadow model. The evaluation used a clear scoring system with four different levels of explanatory quality, ranging from 'exemplary' through 'acceptable' with minor omissions, to 'substandard' with most key points missed, and culminating in 'poor' with essentially no accurate explanation of code functionality. The experts will assign the score from 4 (best) to 1 (worst) respectively.

We selected 50 explanations from each model at random for evaluation by human experts, without disclosing the source of each explanation. The score distribution, as depicted in Figure 11a, primarily shows both models frequently achieving a score of 3, indicating an acceptable level. Notably, both models exhibit a relatively low rate of critical errors, reflected in the 1-point scores. However,

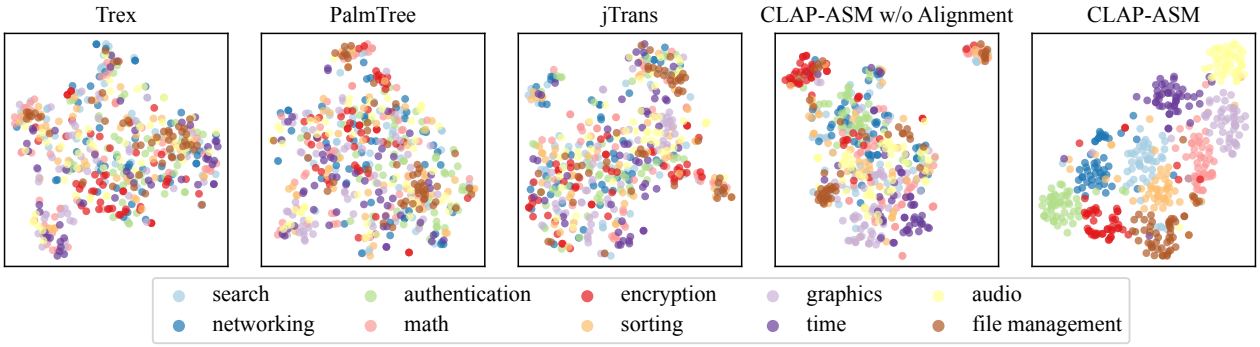


Figure 10: Intrinsic Evaluation: t-SNE visualization of the embedding of assembly code from different models.

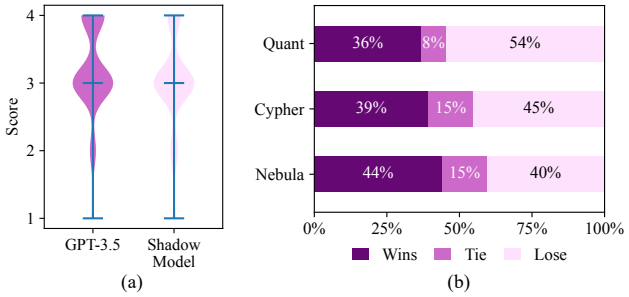


Figure 11: (a) Human evaluation of the explanations of each model by scoring 4 (best) to 1 (worst) individually.
(b) Human evaluation by comparing the explanations from each model side by side, only codename is provided. Win signifies that the user ranks the model above GPT-3.5, while lose implies a lower ranking. A tie means that users perceive both models to have comparable performance. Quant: Vicuña 13B model trained on a 50K dataset. Cypher: Vicuña 13B model trained on a 1.3M dataset. Nebula: LLaMA 30B model trained on a 50K dataset.

GPT-3.5 slightly outperforms with a marginally higher number of 4-point scores, as evidenced by the average scores. GPT-3.5 achieves an average score of 3.14, modestly surpassing the shadow model's average of 3.00. While these results are not outstanding, they are deemed acceptable, suggesting the viability of using language-based supervision as an effective signal for contrastive learning.

Further quantitative analysis via T-Test [59] comparing mean scores of the two models yields a p-value of 0.0036, which indicates that explanations generated by GPT-3.5 are of significantly higher quality than those produced by the shadow model. Noteworthy is that despite the shadow model not achieving parity with GPT-3.5, does not significantly trail in the Section 4.4.2 where the experts can compare the explanations from different models side by side.

4.4.2 Quality of Shadow Model. To compare the quality of source code explanation from the shadow model and GPT-3.5, we employ existing LLM assessment methods [76]. In particular, We design an online website, in which we provide different explanations from various models for the same source code, and let users rank them. This approach enables us to gauge the wins, losses, and ties among the models in comparison to GPT-3.5. During the human evaluation, we aim to prevent model information leakage (specifically, avoid leaking model size and training set size) when obtaining user feedback (assess the output quality of various models).

The final results shown in Figure 11b reveal some discrepancies between GPT-3.5 and the other models in terms of human perception; however, GPT-3.5 does not exhibit significant superiority, as each model experiences wins and losses. Notably, the 30B model, with the highest parameter count, outperforms the two 13B models, aligning with expectations for larger models. Furthermore, the model trained on the 1.3M dataset encounters a wider array of functions, allowing it to tackle more complex tasks, while the model from the 50K training set fares the worst in human evaluations. Nevertheless, it is crucial to stress that the latter model still competes with GPT-3.5 in terms of code comprehension and interpretation. Due to increased GPU memory requirements and reduced batch size associated with the 30B model, along with slower inference speeds from larger models, we opt for the 13B Vicuña model from the 1.3M dataset for extensive inference to strike a balance between performance and inference overhead.

5 BROADER IMPACTS

The experimental results showcased above highlight the notable capabilities of CLAP, yet its potential extends further. By bridging the gap between natural language and assembly code, CLAP facilitates support for zero-shot learning through the use of open vocabulary prompts. This feature enhances CLAP's applicability in more scenarios, offering a wider scope of utility.

In many potential real-world semantic analysis tasks within the realm of binary analysis, the construction of datasets is exceptionally challenging, which has significantly impeded the application of assembly code representation learning. Due to these difficulties in dataset creation, this section analyzes the broader impact of CLAP, through various case studies. These studies not only demonstrate the practical effectiveness of CLAP but also highlight its advantages in zero-shot learning scenarios. This approach showcases the powerful potential of our model in handling complex scenarios where traditional data-driven methods may fall short.

```
    mov edx, 6
.L1:
    xor eax, eax
.L2:
    cmp edx, eax
    jle .L4
    mov ecx, [rdi+rax*4]
    mov esi, [rdi+rax*4+4]
    cmp ecx, esi
    jle .L3
    mov [rdi+rax*4], esi
    mov [rdi+rax*4+4], ecx
.L3:
    inc rax
    jmp .L2
.L4:
    dec edx
    jnz .L1
    retn
```

Figure 12: Assembly code of a bubble sort algorithm

5.1 Fine-grained Assembly Semantics

Our first case study focuses on examining the model's potential comprehension of fine-grained program semantics. We choose sorting algorithms, taking the bubble sort algorithm as an example. We investigate whether the model can recognize the given assembly code as the bubble sort algorithm shown in Figure 12. Our input prompts comprise ten labels, including bubble sort, selection sort, and insertion sort, among others using prompts like "This is a XX sorting algorithm" (replacing XX with various sorting algorithms). Employing a zero-shot inference approach, we search for the prompt that best matches the provided assembly code. The result demonstrates that our model can correctly identify the bubble sort algorithm from the ten options, showcasing its ability to comprehend fine-grained program semantics.

5.2 Malicious Behaviour Classification

In the second case study, we focus on malicious code analysis to demonstrate the model's capabilities in real-world tasks. We manually reverse-engineer functions from malicious samples and find a malicious screenshot function[3]. We construct prompts encompassing common functionalities that may be used in malware, including a screenshot, auto-start, backdoor, download, upload, rootkit, anti-detect, anti-debug, password brute force, and file hijack. We use CLAP to zero-shot differentiate the malicious assembly function from the above categories. The experiment result shows that our model accurately recognizes the function as being related to screenshots, highlighting the practical capability of our model in real-world malicious code analysis.

5.3 Assembly Code Search

We are considering another potential application, which is to assist in reverse engineering by using natural language to search for relevant assembly code which is similar to high-level language code search [12, 35, 42, 67]. Reverse engineers often spend a lot of time analyzing the framework of a program to find its key logic. Using natural language to search for relevant assembly code can greatly improve productivity. For instance, assembly code search could facilitate the detection of potentially malicious code such as unauthorized file downloads and uploads by searching assembly code using natural language queries.

6 DISCUSSION

While our study presents significant advancements with CLAP, we recognize certain limitations and propose directions for future research, along with some lessons we learn while developing CLAP.

[3]Function sub_10001170 in the sample, which can be found at VirusTotal

Combining Source Code for Alignment. Our current approach does not fully harness the source code for training. Future studies could explore methodologies that synergize natural language with source code. This could involve developing more sophisticated algorithms that deeply integrate source code semantics and structures, potentially leading to even more robust models.

Data Source Diversity. Our data generator, relying on compiling packages from Ubuntu and using GPT-3.5 for source code explanations, may introduce biases that could potentially harm generalization ability. Future research could diversify the data sources, including different compilation options or platforms and utilizing varied language models for source code explanations.

Exploring Semantic Analysis Tasks. The current scope of semantic analysis tasks discussed is limited. There is substantial scope for exploring a broader array of application scenarios, such as assembly code search with natural language and function signature recovery. This could include delving into more complex and varied semantic analysis tasks, thereby expanding the model's utility and uncovering new avenues in binary code analysis.

Directly Applying LLM in Tasks. A potential solution for many binary code analysis tasks is directly fine-tuning an LLM. However, LLMs struggle with the complex comprehension required for binary code analysis, a limitation not easily be addressed through straightforward fine-tuning [20, 76]. Moreover, the significant computational demand for LLM is not suitable for embedding tasks which requires a balance between efficiency and effectiveness.

Scaling Beyond Inductive Bias. During the development of CLAP, we observe that by utilizing a model following scaling law [18, 25] (such as Transformer) coupled with a large-scale dataset (e.g., The 195M pairs data used in CLAP), it is possible to achieve commendable outcomes without the need for incorporating various inductive bias and model design from human experts. This observation is also confirmed in GPT [4, 54] and CLIP [53].

7 CONCLUSION

In conclusion, we have presented CLAP, a novel method for learning assembly code representations through natural language supervision. Our approach successfully bridges the gap between assembly code and natural language representations. The CLAP model achieves remarkable transferability in binary analysis and outperforms SOTA solutions in various tasks. This study highlights the potential of learning assembly code representation with natural language supervision, unlocking new possibilities for assembly code analysis and representation learning. We believe that our method opens up new avenues for research and offers a promising starting point for future work.

ACKNOWLEDGEMENTS

We would like to sincerely thank all the reviewers for their valuable feedback that greatly helped us to improve this paper. Additionally, special thanks are extended to Jingwei Yi, Jiyan He, and Bolun Zhang for their invaluable comments and assistance with our experiments. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000) and National Natural Science Foundation of China (61972224).

REFERENCES

[1] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 361–374.

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems* 35 (2022), 23716–23736.

[3] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2021. Improving Language Models by Retrieving from Trillions of Tokens. arXiv:2112.04426 [cs]

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. arXiv:2005.14165

[5] Canonical. 2023. Ubuntu: Enterprise Open Source and Linux. https://ubuntu.com/. Accessed: 2023-06-01.

[6] Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Yanming Zhang, Weiming Zhang, and Nenghai Yu. 2024. FoC: Figure out the Cryptographic Functions in Stripped Binaries with LLMs. arXiv:2403.18403 [cs.CR]

[7] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/

[8] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries.. In *USENIX Security Symposium*. 99–116.

[9] Victor Cochard, Damian Pfammatter, Chi Thang Duong, and Mathias Humbert. 2022. Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 60–73. https://doi.org/10.1109/EuroSP53844.2022.00012

[10] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[11] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. arXiv:2310.03533 [cs.SE]

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/arXiv.2002.08155 arXiv:2002.08155 [cs]

[13] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How Far Have We Gone in Vulnerability Detection Using Large Language Models. arXiv:2311.12420 [cs.AI]

[14] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis.. In *USENIX Security Symposium*. 1787–1804.

[15] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*.

[16] Hex-Rays. 2015. *IDA Pro Disassembler and Debugger*. Retrieved April 10, 2018 from https://www.hex-rays.com/products/ida/index.shtml

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[18] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs]

[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[20] Samyak Jain, Robert Kirk, Ekdeep Singh Lubana, Robert P. Dick, Hidenori Tanaka, Tim Rocktäschel, Edward Grefenstette, and David Krueger. 2024. What happens when you fine-tuning your model? Mechanistic analysis of procedurally generated tasks.. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=A0HKeKl4Nl

[21] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. *arXiv preprint arXiv:2401.11161* (2024).

[22] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova$^+$: Generative Language Models for Binaries. arXiv:2311.13721 [cs.SE]

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[24] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601* (2023).

[25] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. https://doi.org/10.48550/arXiv.2001.08361 arXiv:2001.08361 [cs, stat]

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

[27] Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959* (2018).

[28] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[30] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. 2022. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning*. PMLR, 12888–12900.

[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[32] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251. https://doi.org/10.1145/3460120.3484587 arXiv:2103.03809

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

[34] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *arXiv:1904.12787 [Cs, Stat]*. arXiv:1904.12787 [cs, stat]

[35] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning. https://doi.org/10.48550/arXiv.2308.03281 arXiv:2308.03281 [cs]

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

[37] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. arXiv:2304.08485 [cs]

[38] Huidong Liu, Shaoyuan Xu, Jinmiao Fu, Yang Liu, Ning Xie, Chien-Chih Wang, Bryan Wang, and Yi Sun. 2021. CMA-CLIP: Cross-Modality Attention CLIP for Image-Text Classification. arXiv:2112.03562 [cs]

[39] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/arXiv.1907.11692 arXiv:1907.11692 [cs]

[40] LLVM. 2023. Clang: a C language family frontend for LLVM. https://clang.llvm.org. Accessed: 2023-06-01.

[41] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei,

et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).
[42] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1. Curran. https://doi.org/10.48550/arXiv.2102.04664 arXiv:2102.04664 [cs]
[43] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. [n. d.]. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. ([n. d.]).
[44] Andrea Marcelli, Mariano Graziano, Mohamad Mansouri, Xabier Ugarte-Pedrero, Davide Balzarotti, and Yanick Fratantonio. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. 18.
[45] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*. Internet Society, San Diego, CA. https://doi.org/10.14722/bar.2019.23020
[46] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *arXiv:1811.05296 [Cs]*. arXiv:1811.05296 [cs]
[47] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).
[48] OpenAI. [n. d.]. *GPT-4 Technical Report*. Technical Report.
[49] OpenAI. 2023. ChatGPT. https://chat.openai.com. Accessed: 2023-06-06.
[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.
[51] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2020. XDA: Accurate, Robust Disassembly with Transfer Learning. arXiv:2010.00770 [cs.CR]
[52] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2021. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv:2012.08680 [cs]* (April 2021). arXiv:2012.08680 [cs]
[53] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020
[54] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. [n. d.]. Language Models Are Unsupervised Multitask Learners.
[55] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435* (2017).
[56] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[57] sentence transformers. 2023. Sentence Transformer: MPNet-Base-V2. https://huggingface.co/sentence-transformers/all-mpnet-base-v2/. Accessed: 2023-06-01.
[58] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 611–626.
[59] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.
[60] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. *arXiv preprint arXiv:2401.16185* (2024).
[61] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. arXiv:2403.05286 [cs.PL]
[62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. [n. d.]. LLaMA: Open and Efficient Foundation Language Models. ([n. d.]).
[63] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).
[64] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order Matters: Sequence to Sequence for Sets. *arXiv:1511.06391 [cs, stat]* (Feb. 2016). arXiv:1511.06391 [cs, stat]
[65] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. arXiv:2402.18818 [cs.SE]
[66] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual South Korea, 1–13. https://doi.org/10.1145/3533767.3534367
[67] Hao Wang, Jia Zhang, Yingce Xia, Jiang Bian, Chao Zhang, and Tie-Yan Liu. 2020. COSEA: Convolutional Code Search with Layer-wise Attention. arXiv:2010.09520 [cs.SE]
[68] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs]
[69] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 (Sept. 2021). arXiv:2109.00859 [cs]
[70] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. 363–376.
[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. ACM, Dallas Texas USA, 363–376. https://doi.org/10.1145/3133956.3134018
[72] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: a tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244.
[73] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2709–2725. https://www.usenix.org/conference/usenixsecurity22/presentation/yu-sheng
[74] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (April 2020), 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466
[75] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 3872–3883.
[76] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: Less Is More for Alignment. https://doi.org/10.48550/arXiv.2305.11206 arXiv:2305.11206 [cs]
[77] Wenyu Zhu, Hao Wang, Yuchen Zhou, Jiaming Wang, Zihan Sha, Zeyu Gao, and Chao Zhang. 2023. kTrans: Knowledge-Aware Transformer for Binary Code Embedding. arXiv:2308.12659 [cs.SE]

Received 16-DEC-2023; accepted 2024-03-02