On the Effectiveness of Custom Transformers for Binary Analysis

Xuezixiang Li*, Lian Gao*, Sheng Yu*, Yu Qu†, Heng Yin*
*University of California, Riverside †Xi'an Thermal Power Research Institute Co., Ltd
{xli287, lgao027, syu061}@ucr.edu, quyu@tpri.com.cn, heng@cs.ucr.edu

***Abstract*—In recent years, there has been increasing interest in using deep learning for binary analysis tasks. Particularly, Transformer-based pre-trained language models have attracted enormous attention and obtained encouraging results. Numerous research attempts modified the Transformer network architecture and designed new pre-training tasks explicitly tailored for individual downstream binary analysis tasks, and positive results were reported. However, it remains unclear whether these architectural changes and their associated pre-training tasks are beneficial to other downstream binary analysis tasks, and whether a vanilla Transformer model can perform equally well via fine-tuning.**

In order to provide guidance for future explorations in this direction, in this paper, we evaluate four custom Transformer-based models (i.e. jTrans, PalmTree, StateFormer, and Trex) 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.

***Index Terms*—Deep Learning, Binary Analysis, Language Model, Representation Learning**

I. Introduction

Binary analysis, which encompasses a range of techniques for extracting and inferring information from machine code, has been a critical research area in the field of computer security. In recent years, there has been a growing trend of applying deep learning models to binary analysis tasks such as function boundary identification [1], [2], binary code similarity detection [3], [4], [5], [6], [7], [8], [9], [10], function prototype inference [11], value set analysis [12], vulnerability detection [13], etc. Inspired by the remarkable progress in large language models (LLMs), recent research has demonstrated the efficacy of Transformer-based language models [14] in various binary analysis tasks [5], [15], [16], [10], [17], [18], owing to the shared characteristics of programming languages (PL) including assembly language and natural languages (NL).

This work was done while Yu Qu was a postdoctoral researcher at University of California, Riverside.

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 [16], [10], [5] employ topological features of binary programs, and design pre-training tasks to capture control flow information, while others [17], [18], [15] 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 [10] 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 [17] 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) [19] 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 [10] is designed for function similarity search, and its evaluation was limited to this single task using different baseline models. StateFormer [17] focuses on fine-grained type inference, yet did 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 [16], 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 [20] and ALBERT [21]. 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 their 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 [18]. 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.

II. Background

The Transformer model [14] 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 additional 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.

A. 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 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 1 depicts the architectures of several prominent Transformer-based models. StateFormer introduces a numerical representation module that incorporates the Neural Arithmetic Nunit (NAU) [19], 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 [9] uses the microtrace-based model to generate function token embeddings, but proposed to use a Long Short-Term Memory (LSTM) network to model numerical values involved in arithmetic operations.

jTrans [10] 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 [15] 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 [22].

NeuDep [23] 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 [7] 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

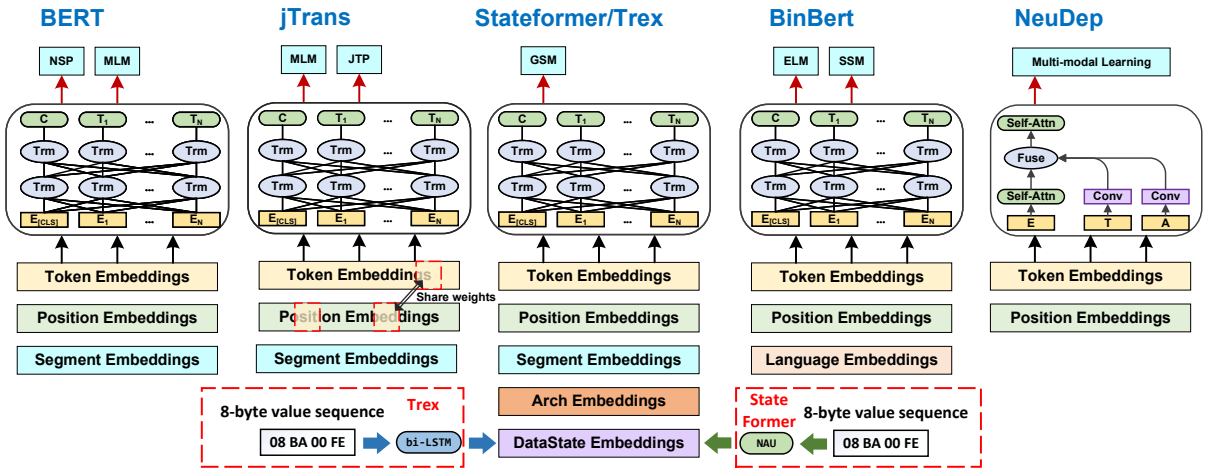


Fig. 1: Architectural Differences among BERT, StateFormer/Trex, jTrans, BinBert, and NeuDep

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. [5] 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 [24] uses DeepSemantic [25] for instruction normalization to alleviate the out-of-vocabulary (OOV) problem. SROBR [26] combines BERT with graph attention networks (GATs) [27] to incorporate control flow features. CodeFormer [28] 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.

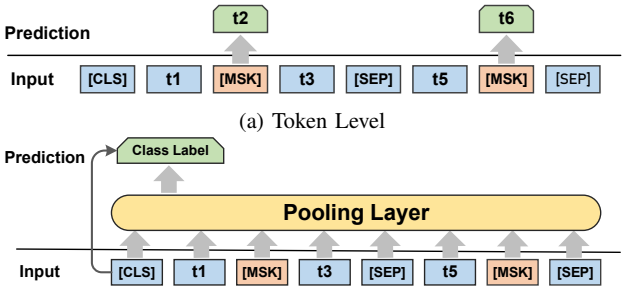
B. Pre-training Tasks

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 2a,) 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 2b) 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 explicitly 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 [10] 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) [17] 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.

In BinBERT [15], 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



(a) Token Level

(b) Sequence Level

Fig. 2: Two types of pre-training tasks

determining whether the given symbolic expression belongs to the set of expressions representative of the strand.

PalmTree [16] 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. [21].

C. 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.

a) ***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 [10]. 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 [29]. 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 [30] ensures that the distance between a negative function pair exceeds a certain margin. The triplet loss [31] 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 [32], 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 [10], 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 [33]. A variant of this downstream task is called Algorithm Classification which is first proposed by TBCNN [34]. More details about the task can be found in §III-E3

b) ***Type Inference****:* Type inference [17], [35], [36], [37] 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.

c) ***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. [18].

d) ***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. [11], 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.

e) ***Other Downstream Tasks****:* In addition to foundational tasks like instruction or function identification, downstream tasks such as Value Set Analysis and Call Graph Recovery pose deeper semantic challenges that test a model's understanding of program behavior.

Value Set Analysis is a static program analysis technique used to determine the possible values that variables or data objects can hold at different points in a program. DEEPVSA by Guo et al. [12] 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, often referred to as Indirect Jump Prediction, poses another significant challenge. 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. [38] demonstrated that this problem can be solved by deep neural networks (DNNs), making it a good candidate to evaluate the models' performance.

While these ML-based methods show promise, they are still in the early stages of exploration. Their effectiveness relative to traditional techniques—such as symbolic execution and heuristics—remains to be thoroughly evaluated in these specific tasks.

III. Evaluation Plan

In this section, we first introduce the models that are evaluated (§III-A), evaluation setup (§III-B), and data preparation (§III-C). Then the evaluations of pre-training tasks are discussed in §III-D and the downstream evaluations are described in §III-E.

A. 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.

According to previous requirements, We collect models shown in Table I 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 [16], 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.

B. 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 II.

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.

C. 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 §III-D and §III-E. 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

TABLE I: 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

TABLE II: Hyperparameters on different sized models

Models	Layers	hidden	# heads	# param	Models	Layers	hidden	# 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

in §III-E3. 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 [18]. Specific details can be found in §III-E4.

D. 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.

1) Generative State Modeling: Generative State Modeling (GSM) is a pre-training task proposed by StateFormer [17] 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 [17], 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,

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

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.

2) Jump Target Prediction: Jump Target Prediction (JTP) is a pre-training task proposed by jTrans [10] 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.

E. 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.

1) 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 [10] 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.

$$\begin{aligned} \text{Recall@}k &= \frac{1}{\mathcal{F}} \sum_{f_i \in \mathcal{F}} \mathcal{I}\big(Rank_{f_i}^{gt} \leq k\big), \\ \mathcal{I}(x) &= \begin{cases} 0, & x = \text{False}, \\ 1, & x = \text{True}. \end{cases} \end{aligned} \tag{2}$$

$$MRR = \frac{1}{\mathcal{F}} \sum_{f_i \in \mathcal{F}} \frac{1}{Rank_{f_i}^{gt}} \tag{3}$$

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 $f_{n1}, f_{n2}, ..., f_{ni}$,

$$loss = -log \frac{e^{sim(f,f_p)/\tau}}{\sum_{N}^{i=1} e^{sim(f,f_{ni})/\tau}} \tag{4}$$

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.

2) 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 [17]. 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 [39], 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 [17], which contains 36 different type labels. A detailed list of types can be found in Table VII in the Appendix.

As mentioned in StateFormer [17], 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 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.

3) 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 [34], [40] 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 [34] for this task. The POJ-104 dataset originates from a pedagogical programming open judge (OJ) system [34] that automates the evaluation of submitted

TABLE III: 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

source code for specific problems by executing the code. As depicted in Table III, 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 [40]. Still, we use the same models as Function Similarity Search in this evaluation.

4) 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 [18], which provides an open-source implementation. Furthermore, SymLM is implemented using the open-source pre-trained model from Trex [9], 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:

$$\mathbb{1}(W, \hat{w}) = \begin{cases} 1, & \hat{w} \in W \\ 0, & \hat{w} \notin W \end{cases} \tag{5}$$

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:

$$tp = \sum_{\hat{w_i} \in \hat{W}} \mathbb{1}(W, \hat{w}), fp = \left\|\hat{W}\right\| - tp, fn = \|W\| - tp, \tag{6}$$

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 §III-E2. Similar to the previous downstream evaluations, we also sampled the testing set 10 times and obtained multiple results to mitigate the randomness.

IV. Evaluation Results

A. Pre-training Tasks

1) Generative State Modeling: Table IV 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.

TABLE IV: 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

2) Jump Target Prediction: As described in §III-D2, we choose BERT, BERT-JTP, and jTrans to evaluate the Jump Target Prediction task. Table IV presents the evaluation results. In the appendix, we also show the accuracy during fine-tuning in Figure 6.

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.

> 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.

B. Downstream Tasks

1) Function Similarity Search: The results of the function similarity search experiments are presented in Figure 3. 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 [10], 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.

2) Type Inference: Table V 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.

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.

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.

3) Algorithm Classification: Figure 4 lists the results of the evaluation on algorithm classification. 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.

4) Function Name Prediction: Table VI shows the results of function name prediction. Similar to §III-E2, 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.

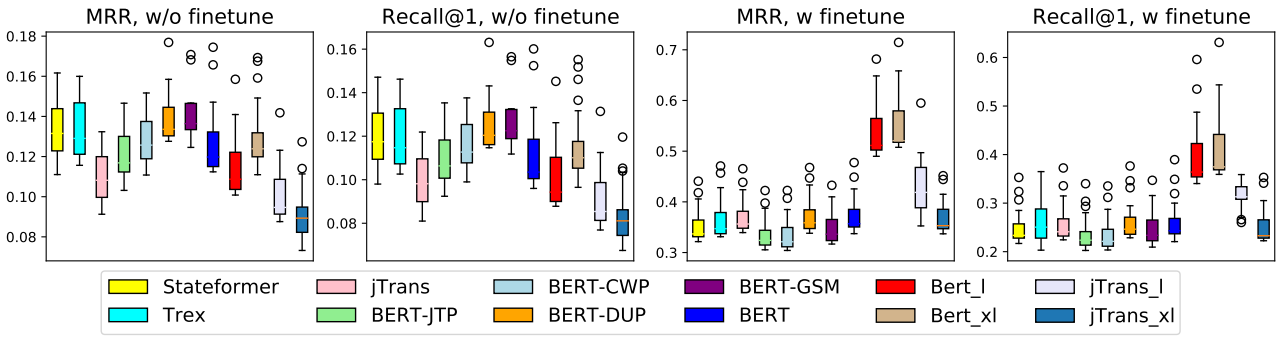


Fig. 3: MRR/Recall@1 of Different model w/ or w/o Fine-Tuning on Function Similarity Search. Pool size = 10000.

TABLE V: Results of Type Inference (Opt-level=Mixed)

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

TABLE VI: Results of Function Name Prediction

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

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.

V. 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

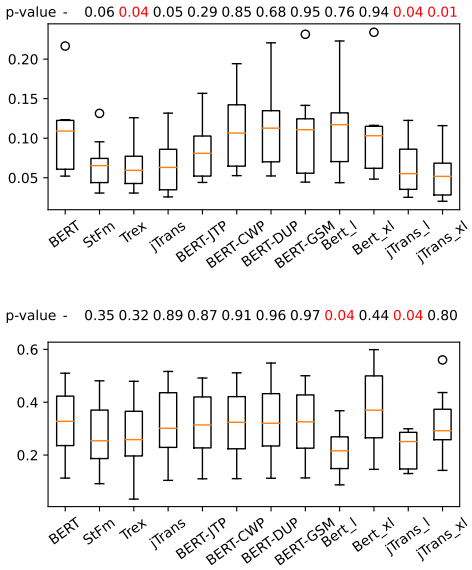


Fig. 4: Results of Algorithm Classification with (top) and without fine-tuining. The p-value shows the T-test results between the target model and BERT. Here, StFm denotes the StateFormer

analysis task may lead to different conclusions.

A. 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.

VI. Related Work

a) ***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 [5], [7], [9], [15], [41], apart from models we evaluate, there exist proprietary models that are not included in our evaluation. OrderMatters [5] 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 [41] utilizes not only assembly code but also source code and related comments for similarity search. BinBert [15] and Trex [9] benefit from dynamic information. BinBert [15] 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 [9] 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 [42] uses the Transformer-based model to match binaries and source code at the intermediate representation (IR) level. SymLM [18] 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 [43] uses BERT to generate embeddings for provenance classification. VulHawk [13] uses a graph neural network along with a Transformer language model to identify vulnerabilities across architectures. Like COMBO [41] and OrderMatters [5], VulHawk [13] 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 [44], [3], [29], [5], [13], [45], [46], [47], [48] try to use Graph Neural Network (GNN) to capture structural features of functions, while SAFE [8] and InnerEye [49] using LSTM with attention mechanism to encode assembly code. The GNN is usually used to encode control flow graphs [44], [3], [29], [5]. It has also been used in disassembly. DeepDi [1] 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 are out of the scope of this work.

b) ***Evaluations on Neural Binary Analysis Approaches****:* Benchmarks play an important role in deep learning-based program analysis research. CodeXGLUE [40] 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 [16] 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 stud-

ies [5], [10], [9], [17], [18] 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. [50] 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.

VII. Conclusion

In this paper, 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.

VIII. Data Availability

Our Dataset, pre-trained model, and code of evaluation will be available at https://github.com/palmtreemodel/transformer-evaluation

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported by Amazon Research Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2709–2725, 2022.

[2] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[3] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and distributed system security symposium*, 2020.

[4] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376, 2017.

[5] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 1145–1152, 2020.

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

[7] Y. Gu, H. Shu, and F. Hu, "Uniasm: Binary code similarity detection without fine-tuning," *arXiv preprint arXiv:2211.01144*, 2022.

[8] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 309–329, Springer, 2019.

[9] 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*, vol. 49, pp. 2776–2790, apr 2023.

[10] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–13, 2022.

[11] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries.," in *USENIX Security Symposium*, pp. 99–116, 2017.

[12] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "Deepvsa: Facilitating value-set analysis with deep learning for postmortem program analysis.," in *USENIX Security Symposium*, pp. 1787–1804, 2019.

[13] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search," *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[15] F. Artuso, M. Mormando, G. A. Di Luna, and L. Querzoni, "Binbert: Binary code understanding with a fine-tunable and execution-aware transformer," *arXiv preprint arXiv:2208.06692*, 2022.

[16] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3236–3251, 2021.

[17] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. 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*, pp. 690–702, 2021.

[18] X. Jin, K. Pei, J. Y. Won, and Z. 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*, pp. 1631–1645, 2022.

[19] A. Madsen and A. R. Johansen, "Neural arithmetic units," in *International Conference on Learning Representations*, 2020.

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

[21] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations*, 2020.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[23] K. Pei, D. She, M. Wang, S. Geng, Z. Xuan, Y. David, J. Yang, S. Jana, and B. 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*, pp. 747–759, 2022.

[24] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, pp. 361–374, 2022.

[25] H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary code representation with bert," *arXiv preprint arXiv:2106.05478*, 2021.

[26] K. Tang, Z. Shan, F. Liu, Y. Huang, R. Sun, M. Qiao, C. Zhang, J. Wang, and H. Gui, "Srobr: Semantic representation of obfuscation-resilient binary code," *Wireless Communications and Mobile Computing*, vol. 2022, 2022.

[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[28] G. Liu, X. Zhou, J. Pang, F. Yue, W. Liu, and J. Wang, "Codeformer: A gnn-nested transformer model for binary code similarity detection," *Electronics*, vol. 12, no. 7, p. 1722, 2023.

[29] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*, pp. 3835–3845, PMLR, 2019.

[30] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, pp. 1735–1742, IEEE, 2006.

[31] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

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

[33] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*, pp. 1597–1607, PMLR, 2020.

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[35] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 99–116, USENIX Association, Aug. 2017.

[36] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. 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*, pp. 152–162, 2018.

[37] A. Maier, H. Gascon, C. Wressnegger, and K. 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*, pp. 288–308, Springer, 2019.

[38] 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)*, (Los Alamitos, CA, USA), pp. 2357–2374, IEEE Computer Society, may 2023.

[39] J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–35, 2016.

[40] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[41] Y. Zhang, C. Huang, Y. Zhang, K. Cao, S. T. Andersen, H. Shao, K. Leach, and Y. Huang, "Combo: Pre-training representations of binary code using contrastive learning," *arXiv preprint arXiv:2210.05102*, 2022.

[42] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 601–612, IEEE, 2022.

[43] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "Binprov: Binary code provenance identification without disassembly," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 350–363, 2022.

[44] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pp. 363–376, 2017.

[45] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3872–3883, 2020.

[46] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[47] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, pp. 1–11, 2019.

[48] G. Kim, S. Hong, M. Franz, and D. 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*, pp. 151–163, 2022.

[49] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. 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.

[50] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2099–2116, 2022.

APPENDIX

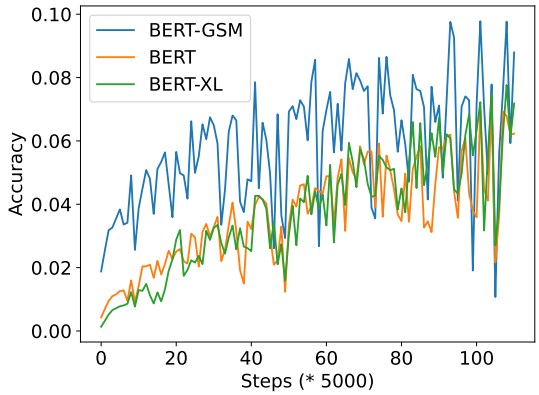


Fig. 5: Accuracy w/o 0 during finetuing: GSM

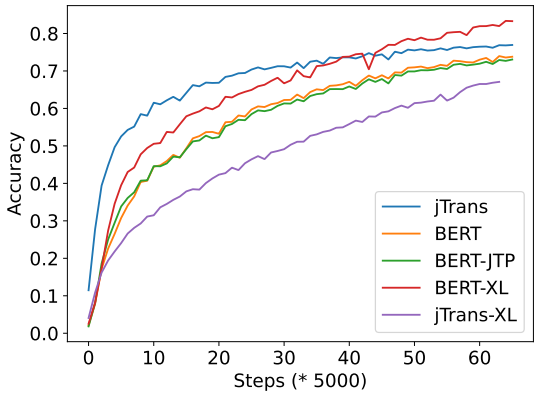


Fig. 6: Accuracy during finetuing: JTP

TABLE VII: 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 *