CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking

ZIAN SU, Purdue University, USA
XIANGZHE XU, Purdue University, USA
ZIYANG HUANG, Purdue University, USA
ZHUO ZHANG, Purdue University, USA
YAPENG YE, Purdue University, USA
JIANJUN HUANG*, Renmin University of China, China
XIANGYU ZHANG, Purdue University, USA

Transformer based code models have impressive performance in many software engineering tasks. However, their effectiveness degrades when symbols are missing or not informative. The reason is that the model may not learn to pay attention to the right correlations/contexts without the help of symbols. We propose a new method to pre-train general code models when symbols are lacking. We observe that in such cases, programs degenerate to something written in a very primitive language. We hence propose to use program analysis to extract contexts a priori (instead of relying on symbols and masked language modeling as in vanilla models). We then leverage a novel attention masking method to only allow the model attending to these contexts, e.g., bi-directional program dependence transitive closures and token co-occurrences. In the meantime, the inherent self-attention mechanism is utilized to learn which of the allowed attentions are more important compared to others. To realize the idea, we enhance the vanilla tokenization and model architecture of a BERT model, construct and utilize attention masks, and introduce a new pre-training algorithm. We pre-train this BERT-like model from scratch, using a dataset of 26 million stripped binary functions with explicit program dependence information extracted by our tool. We apply the model in three downstream tasks: binary similarity, type inference, and malware family classification. Our pre-trained model can improve the SOTAs in these tasks from 53% to 64%, 49% to 60%, and 74% to 94%, respectively. It also substantially outperforms other general pre-training techniques of code understanding models.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Unsupervised learning**.

Additional Key Words and Phrases: Code Language Models, Attention Regularization, Self-supervised Learning

ACM Reference Format:
Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking. *Proc. ACM Softw. Eng.* 1, FSE, Article 26 (July 2024), 24 pages. https://doi.org/10.1145/3643752

*Corresponding author.

Authors' addresses: Zian Su, Purdue University, West Lafayette, USA, su284@purdue.edu; Xiangzhe Xu, Purdue University, West Lafayette, USA, xu1415@purdue.edu; Ziyang Huang, Purdue University, West Lafayette, USA, huan1562@purdue.edu; Zhuo Zhang, Purdue University, West Lafayette, USA, zhan3299@purdue.edu; Yapeng Ye, Purdue University, West Lafayette, USA, ye203@purdue.edu; Jianjun Huang, Renmin University of China, Beijing, China, hjj@ruc.edu.cn; Xiangyu Zhang, Purdue University, West Lafayette, USA, xyzhang@cs.purdue.edu.

ACM 2994-970X/2024/7-ART26
https://doi.org/10.1145/3643752

1 INTRODUCTION

Transformer models [2, 8, 19, 62] have substantially advanced the state-of-the-art of Natural Language Processing (NLP) applications. They are also the technique behind Large Language Models (LLMs) which have demonstrated unprecedented generalization and reasoning ability. These models feature the *self-attention mechanism* [73] that allows them to learn correlations/contexts distant in the input space and self-supervised pre-training methods, such as *Masked Language Modeling* (MLM) that can produce high quality embeddings by masking parts of inputs and forcing the models to predict the masked parts from their contexts. Although Transformer models were initially introduced to improve NLP applications, recent research has shown that they can be used in many software engineering tasks, such as automated program repair [11, 25, 59, 78], software testing [26], vulnerability detection [54, 69] and so on [27, 36, 45, 80], outperforming traditional methods. An underlying reason of their impressive performance is that software has rich natural language artifacts (also called *symbols* in this paper for terminology simplicity), such as comments, documents, variable and function names. These artifacts make programs "understandable" to Transformer models, just like to human developers. For example, the correlation between a statement that defines a variable and another statement that uses the variable can be naturally captured by the attention mechanism due to the common variable name, just like how developers infer dataflow by variable names.

However, when symbols are not available, such as in stripped binary executables, or not informative, such as in obfuscated software, programs become extremely difficult for models to understand, just like they are hard for developers. In particular, they are in a very primitive language in which tokens no longer have rich semantics. For example, in x86 executables, variables are denoted by registers and memory locations dereferenced through registers or constants; a same register may be allocated to multiple variables. As such, the definition of a register and the use of the register may not suggest dependence; neighboring tokens may belong to completely independent contexts/computations. Such context interleavings are difficult to unfold without the help of symbols. As shown by our experiments, code models based on vanilla Transformer architectures and using the vanilla MLM pre-training have degraded performance when symbols are lacking.

To mitigate the problem, researchers have proposed various methods. Trex [57] used micro-executions to acquire input and output operand values of an x86 instruction and then leveraged such values to train a Transformer model to precisely represent instruction level semantics. JTrans [75] used jump target prediction together with MLM to pre-train a Transformer model and used contrastive learning to force the model to learn embeddings that can distinguish similar and dissimilar binary functions. DiEmph [82] further improved JTrans by removing biases (e.g., undesirable code pattern distributions) introduced by compilers. GraphCodeBERT [35] aimed to enhance Transformer code model pre-training leveraging dataflow information. It expanded the raw input sequence with additional tokens denoting variables and introduced extra training losses to force the model to learn data-flow between these tokens. While these proposals demonstrated great improvements over vanilla models, most of them focused on single downstream tasks, instead of general pre-trained models. Some (e.g., GraphCodeBERT) required symbols. Note that there were also a body of works that treated programs as graphs (e.g., control flow graphs and dependence graphs) and leveraged *Graph Neural Networks* (GNNs) for software engineering tasks [23, 31, 41, 71, 84]. However, their performance also degrades when symbols are lacking, e.g., due to their need of labeled data in supervised learning and difficulties in capturing correlations that are multiple edges away in graphs [79]. More can be found in Section 2.

In this paper, we aim to develop a new technique for pre-trained general code models that targets programs without meaningful symbols, binary executables in particular. It does not fine-tune an

existing pre-trained model. Instead, it pre-trains a model from scratch using a new method that regularizes attention. It produces high quality embeddings that encode program dependences and disentangle interleaving contexts and hence enables better performance in downstream tasks. Its intuition is the following. Without meaningful symbols, a programming language degenerates to a very primitive one like an arithmetic language [60], in which the meaning of a variable/statement can only be derived from the direct and transitive computations that produce and use the value of the variable/statement. For example, in a function where all statements in the function body cohesively compute a final return value, the embedding of the return value should reflect the computation of the whole function; a function computing multiple orthogonal output values shall have embeddings reflecting such orthogonality in spite of interleavings of the sub-computations. The language is dissimilar to a natural language, and humans rarely speak in such a primitive fashion. As such, the vanilla MLM method can hardly help the model produce the right attention during pre-training without the help of symbols because MLM mainly leverages the correlations between individual tokens and their left and right contexts (shown in Section 2).

To address the challenge, we propose to use program analysis to derive possible dependences between instructions and then construct *attention masks* from such dependences. The masks enable self-attention between instructions that have dependences and preclude attention among those that are independent. This aligns well with the aforementioned primitive language. The pre-training then helps the model determine which dependences are more important than others and hence deserve more attention. Besides the dependence masks, additional masks are created to explicitly regularize self-attention to correlations other than program dependences, such as token co-occurrences. We also enhance the MLM pre-training by masking part of program dependences and introducing spurious dependences, and then forcing the model to correctly predict/classify such dependences. During inference, CodeArt takes the subject binary and generates the corresponding attention masks and feeds both the input tokens and the masks to produce output embeddings. Note that the dependence analysis and mask construction are deterministic and transparent to users. Our contributions are summarized in the following.

- We propose a new method to pre-train Transformer code models when symbols are lacking. Inspired by existing works that utilize program analysis to enhance Deep Learning models [9, 10, 12, 13, 15, 20–22, 32, 43, 51, 64, 65, 67, 68, 74], our method analyzes program dependences and use them to help self-attention. Different from many existing techniques that focus on improving performance of individual downstream tasks, our pre-trained models are general, serving a large number of applications, and use masks to regulate attention.
- We address a number of technical challenges, including enhancements of tokenization, model architecture, a new pre-training method that masks dependences, transforming transitive dependences to connectivity relations to avoid undesirable decay, and new training objectives. Built on top of the BinaryCorps dataset [75], we construct a large-scale training dataset with 26 million stripped binary functions containing explicit dependence information.
- We develop a prototype CodeArt (*Better CODE models by Attention regulaRizaTion when symbols are lacking*) and use it to pre-train a BERT-like general model from scratch. The pre-training converges in four days with an 8×A100 GPU cluster. To demonstrate the generalization of the pre-trained model, we use it in three downstream tasks: binary similarity analysis, malware classification, and binary type inference. We have improved the SOTAs of these tasks from 53% to 64% (Recall@1 with a pool size of 500), from 49% to 60% (LRAP), and from 74% to 94% (F-1 score averaged over different optimizations), respectively. We empirically compare with other general code pre-training approaches like GraphCodeBERT

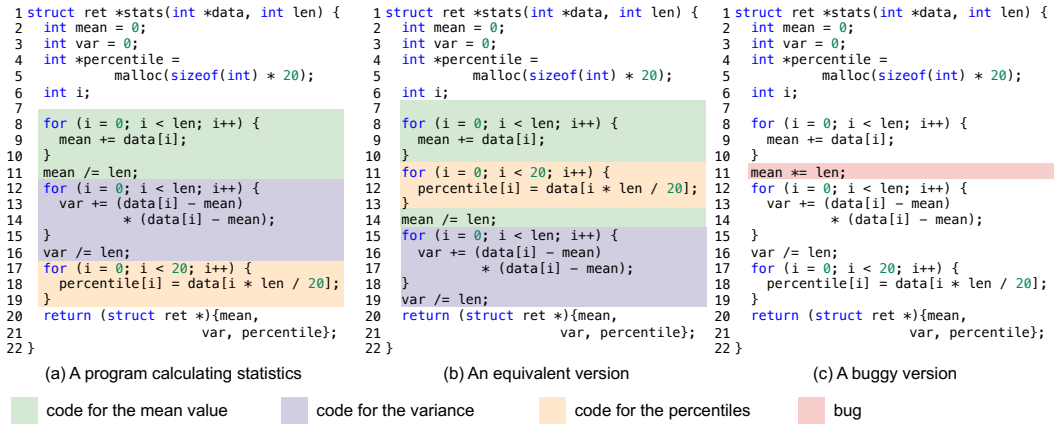


Fig. 1. Code examples calculating statistics: (a) and (b) are equivalent but have different statement orders; (c) is buggy at line 11 with the wrong operation.

on binary code and show that our model is much more effective. We also conduct an ablation study to justify our design choices. [1]

2 MOTIVATION

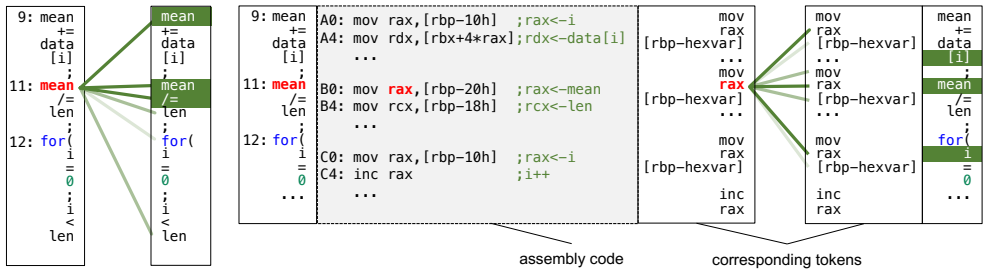
We use an example to discuss the limitations of existing techniques and illustrate our technique. For better readability, we present the example mostly in its source code form. Our technique works on stripped binaries without any symbol information.

Example. Fig. 1(a) shows a code snippet computing mean, variance, and percentile for an array of sorted data. The code blocks in different colors denote the three sub-computations. Fig. 1(b) shows a version equivalent to (a) with statements reordered. Fig. 1(c) is highly similar to (a) except that its line 11 has a buggy operation. We mix (b) and (c) with a set of 5 random functions to form a pool of candidate functions and use (a) to query the most similar function from the pool. We first use a similarity analysis built on the CodeT5 model using source code. The analysis easily identifies that (a) is similar to (b) and not to (c). However, when we use a binary similarity analysis JTrans that operates on stripped binaries (without symbols), JTrans mistakenly considers (a) is similar to (c) instead of (b). In contrast, a similarity analysis built on our model reports the correct similarity result even without symbols. In the following, we explain these different results.

Transformer Models. Recent research has shown that Transformer based code models, including Large Language Models (LLMs), deliver superior performance in many software engineering tasks [29, 48, 63, 77]. Transformer was initially introduced for NLP applications [19, 61, 62, 73], and the rich natural language artifacts in software make it suitable for these tasks. However, in tasks where symbols are precluded (e.g., binary analysis) or not informative (e.g., in obfuscated software), code becomes dissimilar to natural language products, causing the model to have sub-optimal attention and hence performance degradation.

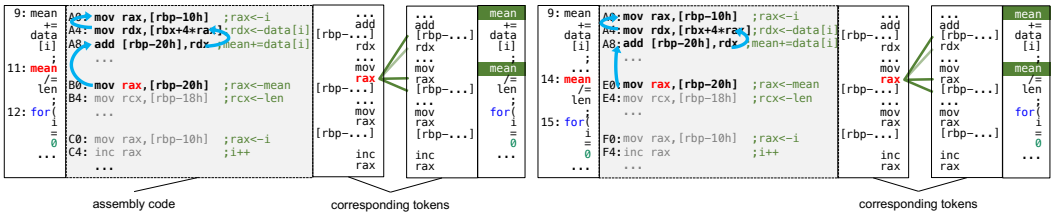
Fig. 2(a) shows part of the attention map of lines 9–12 in the code in Fig. 1(a), by the CodeT5 model that works on source code. Observe that with the help of variable name `mean`, the model correctly correlates statements with program dependences, e.g., the strong attention between *mean* at line 11 and *mean* at line 9. In contrast, Fig. 2(b) shows the attention map of the corresponding binary code by the JTrans model [75]. Here, `rax@A0` denotes variable `i`, `rax@B0` denotes `mean`

[1] Code and data are available at https://github.com/ziansu/codeart. Reproduction package at [1].



(a) Attention map on source code (b) Attention map on binary code

Fig. 2. Attention maps of variable mean at line 11 of Fig. 1(a) (in red), by a source code model (a) and by a binary code model (b). In (a), we show attention between source code tokens, corresponding to lines 9–11 in Fig. 1(a). In (b), we show attention between binary code tokens. For readability, we also include the corresponding source code and part of the assembly. The variable *mean* is stored in the register rax at line B0 (in red). A line with a darker color denotes a larger attention value. In an instruction, the first operand is destination and the second the source, and comment "rax<-i" means that register rax stores variable i.



(a) Attention map of CodeArt (b) Attention map of CodeArt on the equivalent version

Fig. 3. Attention maps by CodeArt for variable mean in the code snippets in Fig. 1 (a) and (b). The instructions in bold are in the dependence context of the variable and the blue arrows denote dependences.

and rax@C0 denotes i. Observe that the model has undesirable (strong) attention among the rax's. Moreover, as the binaries for the programs in Fig. 1(a) and (c) are syntactically very similar. The JTrans model produced highly similar (but problematic) attention maps for the two, causing its misclassification of the two as similar functions. In Section 4.1, our results show that CodeArt outperforms the JTrans model on the binary similarity task by over 30% in zero-shot settings.

GNN Models. GNNs are also widely used in software engineering tasks [4, 7, 17, 37, 44]. The basic idea is that programs have explicit graph structures, such as control flow graph (CFG), abstract syntax tree (AST), and program dependence graph (PDG), which can be leveraged by GNNs. In GNNs, the embedding of a node (e.g., a basic block in CFG) can be derived from the embeddings of neighboring nodes and the content of the node itself. They are hence a plausible solution when symbols are missing. However, as pointed out in [75, 79], standard GNN based code models struggle to capture long-range dependences [86]. Although scaling GNNs in depth and width may mitigate the problem, it may in the meantime cause optimization instabilities and representation oversmoothing [6, 14, 42]. Our experiments in Section 4.1 show that CodeArt outperforms GNN based code models on the binary similarity task by over 20%.

Our Method. Our key observation is that when symbols are lost or not informative, code becomes dissimilar to natural language products, and exhibits its own characteristics. First, *the semantics of a statement is the aggregation of all the statements that directly or transitively contribute to it and those that it contributes to.* For example, the meaning of variable mean at line 11 in Fig. 1(a) is determined

by lines 8-11 and lines 13-14, which form a context for `mean`. In an extreme case where all statements in a function are devoted to computing a final return value, the embedding of the value shall reflect the meaning of the whole function. This is quite different from how humans use natural languages. Second, *absolute code positions shall be de-emphasized when meaningful symbol information is not present to help disentangle interleaving contexts*. Program code tends to have interleaving contexts, which may not be a problem when symbols are present to help disentanglement. For example in Fig. 1(b), the context of `mean` interleaves with the context for computing percentile, namely, lines 11-13. Although it may not be a problem when variable names are present to disentangle the contexts, it becomes a lot more challenging when symbols are lost.

Based on the observations, we propose a new technique that is based on Transformer, and enhanced with a new pre-training method to explicitly regulate self-attention using masks. In particular, it uses program analysis to determine possible dependences between instructions, including both data and control dependences. The masks ensure that a token can only pay attention to its *dependence context*, which includes the instructions that it directly or transitively depends on and those that are directly or transitively dependent on the token. As we will show in Section 3.4, additional masks are also derived to regulate attention in other types of contexts such as *instruction local contexts* that only allow an token to pay attention to other tokens within the same instruction and *global contexts* that model token co-occurrences (within and across instructions). The pre-training further helps the model learn which of these allowed attentions ought to be strong. As shown in Fig. 3(a), the attention map by CodeArt for the binary version of Fig. 1(a) closely resembles that when the source code is used (Fig. 2(a)). Moreover, the attention map by CodeArt for the binary version of Fig. 1(b) (i.e., the reordered but equivalent version) is also highly similar (see Fig. 3(b)), despite their syntactic differences. In fact, CodeArt produces similar attention maps for all three code snippets in Fig. 1. However, the different operations at line 11 of Fig. 1 (a) and (c) cause different final embeddings, allowing a binary similarity analysis built on our model to correctly recognize the similar functions.

Comparison with GraphCodeBERT. GraphCodeBERT [35] is a source code representation model based on BERT architecture that leverages data-flow information in pre-training. By augmenting (appending) the [comment, source code] input with additional variable tokens, and forcing the variable tokens to align with their corresponding tokens in the source code part of the input, and to have the intended data-flow, GraphCodeBERT aims to have better embeddings. Although GraphCodeBERT has demonstrated advantages over the vanilla CodeBERT [29], porting it to handling binaries without symbols is challenging. In particular, its variable token alignment largely relies on the variable name equivalence. At the binary level, the multiple occurrences of a variable may have completely different register/memory tokens, rendering the alignment much more difficult. Without proper alignment, the data-flow training is infeasible. This is supported by our evaluation results of a binary version of GraphCodeBERT in Section 4.5.

3 DESIGN

Overview. Fig. 4 shows the pipeline of CodeArt encoder. Given a binary executable, CodeArt first disassembles the code (step (a)). It then tokenizes the disassembly (step (b)) and performs dependence analysis to derive both data and control dependences (step (c)). In (d), the dependence transitive closures are computed for individual instructions, by traversing dependence edges in both forward and backward directions. The closures are further transformed to connectivity graphs in which an edge is introduced between two nodes if one is reachable from the other. In *Mask Builder* (e), the connectivity graph and the instruction tokens are leveraged to construct an attention mask and a relative distance matrix that measures the dependence distance between two connected

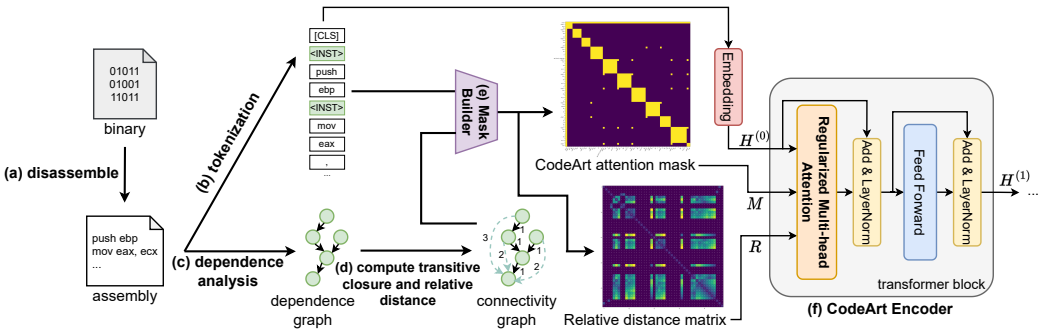


Fig. 4. CodeArt encoder pipeline

nodes. They are further fed to a BERT-like Transformer model (step (f)) to produce the output embedding. Our model is different from the vanilla BERT as it leverages the mask and the distance matrix to regulate attention. The pre-training of CodeArt model entails masking input tokens and masking/perturbing the connectivity graph. In the following subsections, we introduce the individual components.

3.1 Tokenization

CodeArt directly works on binary executables with symbols stripped. Given an executable, CodeArt first disassembles it using IDA-Pro [2]. The disassembled code is then tokenized. The tokenizer breaks each (x86-64) instruction to multiple tokens. Specifically, opcode and operands are denoted by separate tokens. A compound operand may be further broken down to multiple tokens. For example, a memory read instruction `mov rdx,[rbx+4*rax]` that reads a value from an address denoted by `rbx+4*rax` to `rdx` is tokenized to a sequence `'mov', 'rdx', ',', '[', 'rbx', '+', '4', '*', 'rax', ']'`. In addition, we use a special delimiter token `<INST>` to denote the beginning of an instruction. This is critical to the attention regularization which we will discuss in Section 3.4. Formally, a tokenized sequence can be described as $A = \{\mathtt{i}_1, a_{1,1}, \cdots, a_{1,n_1}, \mathtt{i}_2, a_{2,1}, \cdots, a_{2,n_2}, \cdots\}$, where i_t denotes the delimiter for the t-th instruction and $a_{t,n}$ the n-th token of the t-th instruction. Similar to other BERT-based models [19, 29, 47], we prepend a `[CLS]` token to the token sequence. The final input sequence is hence $\boldsymbol{X} = [\mathtt{[CLS]}, A]$.

3.2 Dependence Analysis

Given binary executables, CodeArt first uses program analysis to determine both control dependences and data dependences between instructions, and then uses such information to regulate attention during training and inference. In particular, CodeArt employs IDA Pro [38] to construct the control flow graph (CFG) for each function. Using these CFGs, we resort to a conventional algorithm [30] to determine control dependences. We additionally tailor a source-code data-flow analysis [5] to facilitate the analysis of data dependences in binaries. This data-flow analysis begins by gathering a collection of variables accessed by each statement (or, in the context of binary analysis, an instruction) and subsequently determines the def-use relationship among these variables. Precisely identifying variables accessed by a binary instruction is notably challenging [85], given that all variables are compiled into plain registers and memory locations without any symbol information. To this end, we adopt an approach that overestimates the memory regions an expression

[2] https://hex-rays.com/ida-pro/

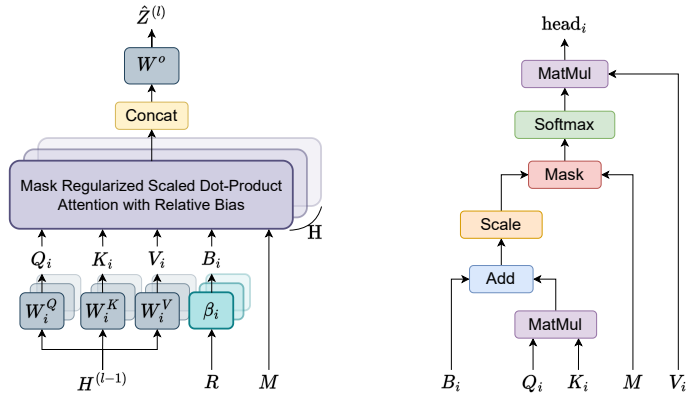


Fig. 5. Regularized Multi-head Attention (with (b) illustrating a zoom-in view of part of (a))

can potentially reference (and hence, the variables that an instruction with the expression can potentially access), shown as follows.

- Expressions denoting stack memory addresses with statically decidable offsets (e.g., `[rsp + 0x20]`) are interpreted to merely reference the corresponding stack locations.
- Expressions denoting stack memory addresses with statically undecidable offsets (e.g., `[rsp + rax * 8 + 0x30]` with `rax` holding an input parameter) are presumed to reference the entire stack frame of the corresponding function.
- Expressions pointing to addresses not statically associated with stack pointers (e.g., `[rbx + rax]` denoting some heap access) are conservatively considered to reference the entire memory space of the given binary.

For instance, given the assembly code: "`(a) mov rax, rbx; (b) mov [rsp + 0x10], rax; (c) mov rcx, [rsp + 0x10]; (d) mov rdx, [rcx];`", our analysis can derive three distinct data dependences from it, including `(a)`←`(b)` through the register `rax`, `(b)`←`(c)` involving the stack memory address `[rsp + 0x10]`, and `(b)`←`(d)` related to the non-stack memory address `[rcx]`. Note that `[rcx]` is assumed to possibly access the entire memory space, which encompasses `[rsp + 0x10]`. It is worth noting that our dependence analysis is largely standard, and we include it for completeness. We do not claim contributions on the analysis.

3.3 Model Architecture

Fig. 4(f) shows our model architecture, which is a multi-layer Transformer encoder [73], with the component *Regularized Multi-head Attention* (RMA) containing the main differences from a standard BERT architecture [19, 47]. As shown in the figure, the encoder takes a sequence $\boldsymbol{X}$ of N tokens and acquires a sequence of input vectors $\boldsymbol{H}^{(0)} = \left[\boldsymbol{h}_1^{(0)}, \cdots, \boldsymbol{h}_N^{(0)}\right] \in \mathbb{R}^{N \times d_h}$ by summing the token embeddings and the corresponding trainable absolute position embeddings (the sum operation is not shown in the figure for brevity). Here, d_h is the dimension of hidden states. The output embedding is then obtained by applying L layers of transformer blocks, each regulated by the CodeArt attention mask $\boldsymbol{M} \in \mathbb{R}^{N \times N}$ and the relative distance matrix $\boldsymbol{R} \in \mathbb{N}^{N \times N}$, as exampled by the block in Fig. 4(f). Formally, the hidden states after layer l are denoted as $\boldsymbol{H}^{(l)} = \text{transformer\_block}\left(\boldsymbol{H}^{(l-1)}, \boldsymbol{M}, \boldsymbol{R}\right),\ l \in \{1..L\}$. A transformer block (Fig. 4 (f)) is formally

defined as follows.

$$\hat{\boldsymbol{Z}}^{(l)} = \text{RMA}(\boldsymbol{H}^{(l-1)}, \boldsymbol{M}, \boldsymbol{R}), \quad \boldsymbol{Z}^{(l)} = \text{LN}(\hat{\boldsymbol{Z}}^{(l)} + \boldsymbol{H}^{(l-1)}), \quad \boldsymbol{H}^{(l)} = \text{LN}(\text{FFN}(\boldsymbol{Z}^{(l)}) + \boldsymbol{Z}^{(l)}) \tag{1}$$

Here RMA is an enhanced multi-head self-attention module [73], FFN is a two-layer feed-forward network, and LN is layer normalization. Their details are shown in Fig. 5. In particular, for each layer l, the output $\hat{Z}^{(l)}$ of RMA is computed as follows. We omit the layer annotation for simplicity.

$$\boldsymbol{Q}_i = \boldsymbol{H}\boldsymbol{W}_i^Q, \quad \boldsymbol{K}_i = \boldsymbol{H}\boldsymbol{W}_i^K, \quad \boldsymbol{V}_i = \boldsymbol{H}\boldsymbol{W}_i^V \tag{2}$$

$$\text{head}_i = \text{softmax}\left(\frac{\boldsymbol{Q}_i\boldsymbol{K}_i^T + \boldsymbol{B}_i}{\sqrt{d_k}} + \boldsymbol{M}\right)\boldsymbol{V}_i \tag{3}$$

$$\hat{\boldsymbol{Z}} = [\text{head}_1; \cdots; \text{head}_\text{H}]\,\boldsymbol{W}^o \tag{4}$$

Here the $(l-1)$-th layer's output $\boldsymbol{H}^{(l-1)} \in \mathbb{R}^{N \times d_h}$ is converted to queries $\boldsymbol{Q}_i = \left[\boldsymbol{q}_{i,1}, \cdots, \boldsymbol{q}_{i,N}\right]$, keys $\boldsymbol{K}_i = \left[\boldsymbol{k}_{i,1}, \cdots, \boldsymbol{k}_{i,N}\right]$ and values $\boldsymbol{V}_i = \left[\boldsymbol{v}_{i,1}, \cdots, \boldsymbol{v}_{i,N}\right]$ for head i, by linear projections with weights $\boldsymbol{W}_i^Q$, $\boldsymbol{W}_i^K$, and $\boldsymbol{W}_i^V \in \mathbb{R}^{d_h \times d_k}$. H is the number of heads, d_k the head dimension, and $\boldsymbol{W}^o$ the weight for the linear projection after concatenating multiple head outputs. $\boldsymbol{M}$ denotes a main difference between our architecture and vanilla Transformer models. It is the *multi-head attention mask*, where $\boldsymbol{M}_{uv}$ is set to either 0 or $-\infty$ to determine whether token u is allowed to attend to token v or not, respectively. Intuitively, by setting parts of the mask to 0, we only allow the model to learn self-attention for the corresponding token pairs and preclude spurious attention. We will further explain how to construct the mask later in the section.

As shown in Equation 3 and Fig. 5(b), another main difference between RMA and standard attention is that RMA incorporates relative positional embedding $\boldsymbol{B}_i$ to explicitly encode position relationship between <INST> delimiter tokens (not between other tokens internal to an instruction). Intuitively, it denotes the dependence distances between instructions as their absolute positions/distances could be misleading when symbols are missing. The relative positional embedding is integrated in the form of bias in Equation 3. It is computed as follows.

$$\boldsymbol{B}_{i,uv} = \beta_i\,(\min(\boldsymbol{R}_{uv}, r_{\max})) \tag{5}$$

Here $\boldsymbol{R}$ is the relative distance matrix pre-computed by our analysis (details in the next section), and β_i is the embedding function that maps a distance value to a trainable parameter for head i. The relative position relationship is learned through these parameters during training. Variable $r_{\max}$ is the max relative distance. Since larger relative distances are less frequent in practice, the model can hardly learn the relative bias for large relative distances. The max distance ensures that when the distance is too large, the bias becomes indistinguishable to that of at distance $r_{\max}$. As such, the model understands that the distance is large and potentially out of distribution.

3.4 Attention Regularization by Masking

In this section, we explain how we construct the attention mask $\boldsymbol{M}$. As discussed earlier, without symbols, model training tends to determine contexts by names of primitive operands (e.g., register names) and absolute positions, which could be misleading. We leverage masking to direct the self-attention to the right places such that correct contexts can be extracted. Fig. 6 provides a conceptual illustration. In (a), a small program consisting of six instructions is shown in the first row (and in their source code form for readability), and each is broken down to multiple tokens. For example in the second row, the green blocks with numbers show the <INST> tokens, each of which is followed by the (grey) tokens of the corresponding instruction. The entire sequence is preceded by a [CLS] token. Assume we are interested in the embedding of the second token of instruction 5 (the 'edx' token in red). First of all, we want the model to pay attention to the co-occurrences

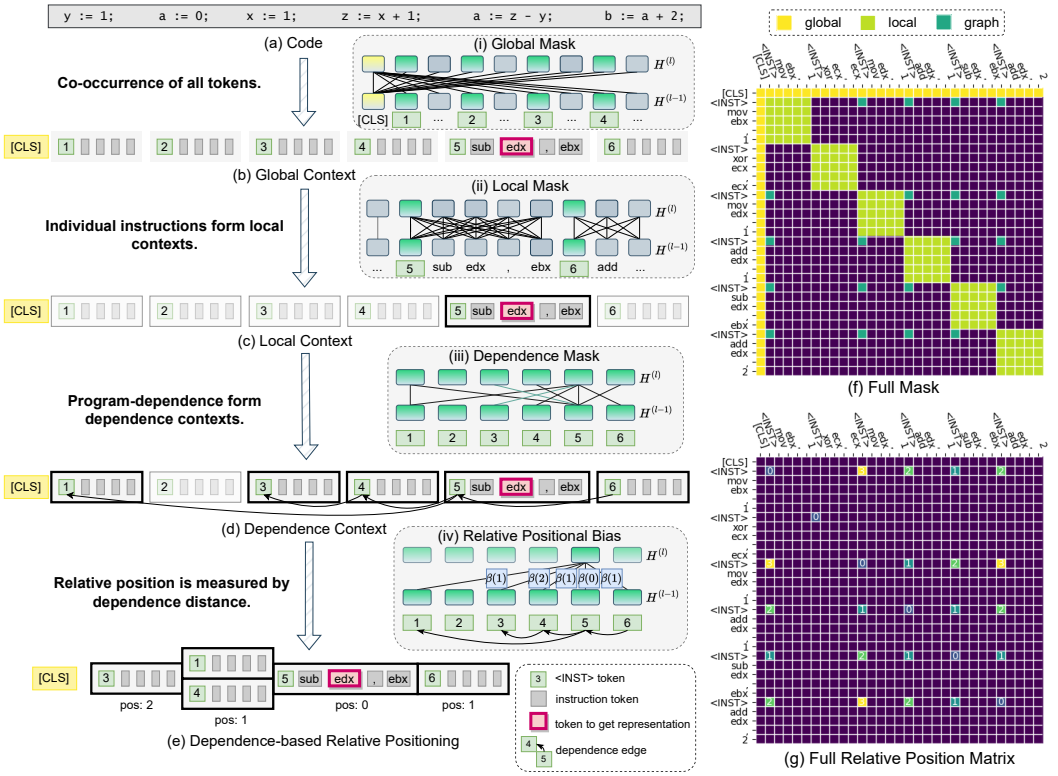


Fig. 6. Contexts, attention masks and relative position matrix

of all the tokens, without focusing on any specific ones. We call it the *global context* (Fig. 6(b)). Second, the model should pay attention to all the tokens in the same instruction as the red token when encoding it. We call it the *local context* of the red token (Fig. 6(c)). Additionally, the model should pay attention to the sub-computation which the red token is in, namely, instructions 1, 3, 4, and 6 due to program dependences. We call it the *dependence context* (Fig. 6(d)). Finally, the neighboring tokens of the red token should be measured by their dependence distances (i.e., how many dependence edges away) instead of their absolute positions. We call it the *dependence based relative positioning* (Fig. 6(e)). Intuitively, it rearranges the instructions by their dependence distances. For example, instruction 5 is closer to instructions 1, 4, 6 than instruction 3.

In standard Transformer models, the self-attention (without masking) allows a query $\boldsymbol{q}_u \in \mathbb{R}^{1\times d_k}$ (an element of $\boldsymbol{Q}$ in Equation 2, corresponding to a token) to attend to individual keys $\boldsymbol{k}_v \in \mathbb{R}^{1\times d_k}$ $(v \in \{1..N\})$ (an element of $\boldsymbol{K}$ in Equation 2) by computing a scaled dot-product $\boldsymbol{q}_u \cdot \boldsymbol{k}_v^T / \sqrt{d_k}$ as the attention score, which is further normalized by a softmax function to get attention weights o_{uv} for aggregating values $\boldsymbol{v}_v \in \mathbb{R}^{1\times d_k}$ $(v \in \{1..N\})$ by $\boldsymbol{z}_u = \sum_{v=1}^{N} o_{uv}\boldsymbol{v}_v$ (similar to Equation 3). The aforementioned multi-step context extraction is achieved by *attention regularization* using masking. As shown in Equation 3, an attention mask $\boldsymbol{M}$ is added to the dot-product before the softmax, which contains a 0 value (to enable attention) or a $-\infty$ value (to disable attention). Note that adding $-\infty$ to the dot-product ensures that the attention weight o_{uv} after the softmax becomes 0 so that the information cannot be aggregated from value $\boldsymbol{v}_v$ into the hidden state. When the mask value is 0, the attention weight is the same as the default one, allowing self-attention. $\boldsymbol{M}$ consists of three kinds of masks, corresponding to the aforementioned global, local, and dependence contexts.

Global Attention Mask $M^{\mathbf{Gl}}$. As shown in Fig. 6(i) in between the first and the second rows on the left, the yellow `[CLS]` token is used to facilitate learning the global context. Observe that it attends to all the tokens, and vice versa. This allows the model to learn *co-occurrences*. For example, the tokens in two separate instructions could indirectly attend to each other through the `[CLS]` token. The yellow cells in Fig. 6(f) show the 0 values in $\boldsymbol{M}^{\mathrm{Gl}}$ of our small program in the form of heat-map, with the legends the tokens and the black cells denoting $-\infty$. Formally,

$$M^{\mathrm{Gl}}_{i,j} = \begin{cases} 0, & X[i] = \texttt{[CLS]} \vee X[j] = \texttt{[CLS]}, \\ -\infty, & \text{otherwise.} \end{cases} \tag{6}$$

Local Attention Mask $M^{\mathbf{Lo}}$. $\boldsymbol{M}^{\mathrm{Lo}}$ ensures a token inside an instruction has attention to all tokens in the same instruction. Each instruction is broken down to an `<INST>` delimiter followed by a sequence of instruction tokens denoting opcode and operands (e.g., the 5th instruction in Fig. 6(b)). As shown in Fig. 6(ii), each token in the 5th instruction attends to all the tokens within the instruction. Moreover, any cross-instruction attention is forbidden in this mask. The green cells in Fig. 6(f) show the 0 values in $\boldsymbol{M}^{\mathrm{Lo}}$ of our small program. Formally,

$$M^{\mathrm{Lo}}_{i,j} = \begin{cases} 0, & \texttt{instruction\_of}(X[i]) = \texttt{instruction\_of}(X[j]), \\ -\infty, & \text{otherwise.} \end{cases} \tag{7}$$

Dependence Attention Mask $M^{\mathbf{Dep}}$. A straightforward method to construct the dependence mask is to directly reflect the directed program dependency graph $G_{\mathrm{dep}} = (V, E_{\mathrm{dep}})$ obtained by our dependency analysis through the `<INST>` tokens, namely, if an instruction A is dependent on another instruction B (i.e., there is a dependence edge $A \longrightarrow B$, we allow A's `<INST>` token to pay attention to B's). However, such a simple design suffers two problems: (1) one layer can only pass information from 1-hop neighbors such that signals from multi-hop neighbors become undesirably weak. (2) the dependence attention is uni-directional whereas bidirectional attention is proved to be better in Transformers pre-training [19].

Thus, we use the Floyd-Warshall algorithm [18] to transform the dependence graph to a connectivity graph $G_{\mathrm{con}} = (V, E_{\mathrm{con}}, \mathcal{D})$, where an undirected edge between two nodes denotes if one is reachable from the other in the original graph, and $\mathcal{D} : V \times V \mapsto \mathbb{N}$ is the distance function which maps a connectivity edge to its path length in the original graph. Then, we construct the mask to enable bidirectional attention if a connectivity edge exists. Fig. 6(iii) shows the dependence mask. Observe that since instruction 5 is dependent on instructions 1 and 4, transitively dependent on 3, and 6 is dependent on 5, symmetric attention is allowed between 5 and 1, 3, 4, 5, and 6. The blue cells in Fig. 6(iii) show the 0 values in $\boldsymbol{M}^{\mathrm{Dep}}$ of our small example. Note that although we only allow direct attention between `<INST>` tokens, tokens internal to instructions attend to `<INST>` and vice versa such that individual internal tokens can attend to other internal tokens in the dependence context through an additional layer of information propagation. Formally,

$$M^{\mathrm{Dep}}_{i,j} = \begin{cases} 0, & X[i] = \texttt{<INST>} \wedge X[j] = \texttt{<INST>} \\ & \wedge \Big(\texttt{instruction\_of}(X[i]) \ \textbf{trans-deps} \ \texttt{instruction\_of}(X[j]) \\ & \vee \texttt{instruction\_of}(X[j]) \ \textbf{trans-deps} \ \texttt{instruction\_of}(X[i]) \Big), \\ -\infty, & \text{otherwise.} \end{cases} \tag{8}$$

The final mask $\boldsymbol{M}$ is the union of above three masks, as in Fig. 6(f). The distance function D of the connectivity graph is further used to construct the relative distance $\boldsymbol{R}$ in Equation 5. Fig. 6(iv)

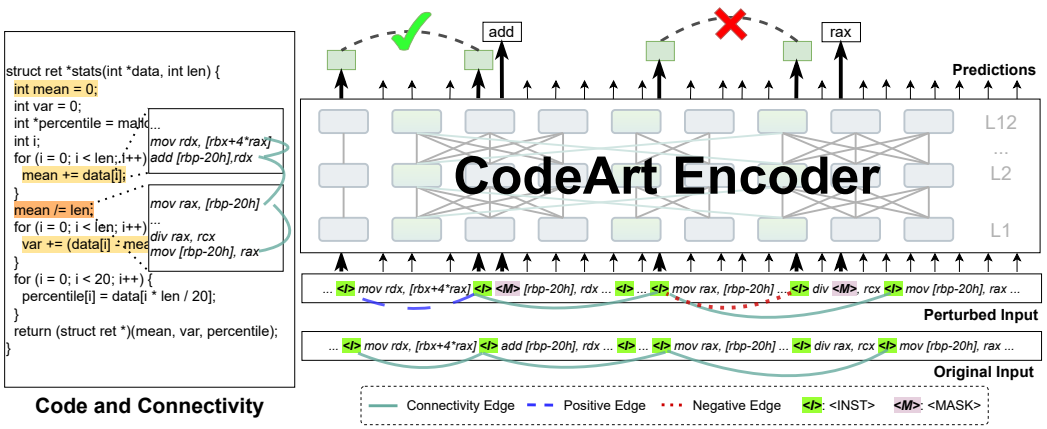


Fig. 7. Pre-training in CodeArt

shows the relative positional biases for the 5th instruction (i.e., the annotations on attention edges). Fig. 6(g) shows the constructed $\boldsymbol{R}$ with different colors denoting different distances.

3.5 Model Pre-training

The pre-training of CodeArt consists of two parts: the traditional *Masked Language Modeling* (MLM) that masks parts of input tokens and *Masked Dependence Modeling* (MDM) that masks existing dependence edges and introduces spurious dependence edges. The essence of MDM is that well-trained contextual embeddings shall be able to predict existing dependence edges (called *positive edges*) and preclude spurious edges (called *negative edges*).

Masked Language Modeling. We perturb the input sequence X following the MLM in RoBERTa [47]. In particular, we sample 15% of the tokens in X. We then replace them with a `[MASK]` 80% of time, with a random token 10% of time, and have them unchanged in the remaining 10%. During pre-training, the model is supposed to predict the masked tokens. Let $\tilde{X}$ be the perturbed sequence. The MLM training loss is formally defined as follows.

$$\mathcal{L}_{\text{MLM}} = \sum_{x \in X^{\text{masked}}} -\log P\left(x \middle| \tilde{X}, \tilde{\boldsymbol{M}}, \tilde{\boldsymbol{R}}\right) \tag{9}$$

where $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{R}}$ are the perturbed attention mask and relative position matrix respectively, which we will discuss next.

Masked Dependence Modeling. In each pre-training step, we randomly sample 40% of the nodes in the connectivity graph $G_{\text{con}} = (V, E_{\text{con}})$. The sampled set is denoted as $V_s \subset V$. Let $E_{\text{con}}(V_s)$ be the set of edges in the connectivity graph that have at least one node in V_s. It denotes the sampled positive edges. We then sample an equal number of pair-wise relations in $V_s \times V \cup V \times V_s - E_{\text{con}}$, which denote the negative edges. These two edge sets form a balance set E_b of positive and negative samples. We then force the model to learn to correctly classify the positive and negative edges during pre-training. Let $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{R}}$ be the perturbed attention mask and relative matrix after the sampled positive edges removed from G_{con} and the negative edges added, and $\boldsymbol{h}_u, \boldsymbol{h}_v$ the hidden states of u and v. The training loss is formally defined as follows.

$$\mathcal{L}_{\text{EdgePred}} = \sum_{e_{uv} \in E_b} -\mathbb{I}(e_{uv} \in E_{\text{con}}) \log p_{e_{uv}} - \mathbb{I}(e_{uv} \notin E_{\text{con}}) \log(1 - p_{e_{uv}}) \tag{10}$$

where

$$p_{e_{uv}} = P\left(e_{uv} \middle| \tilde{X}, \tilde{\boldsymbol{M}}, \tilde{\boldsymbol{R}}\right) = \text{sigmoid}(\boldsymbol{h}_u \cdot \boldsymbol{h}_v^T) \tag{11}$$

Note that even though MDM is similar to GraphCodeBERT's edge prediction and node alignment losses [35] as they are all in the form of binary cross-entropy, there is a core difference that MDM predicts edges between <INST> tokens instead of variable tokens from original code as in GraphCodeBERT. This avoids the risk that edge prediction loss may affect the representation of variable tokens learned by the MLM loss in an undesirable way.

The final loss is the sum of $\mathcal{L}_{\text{MLM}}$ and $\mathcal{L}_{\text{EdgePred}}$.

Example. Fig. 7 presents how we use the motivation example in pre-training. The left shows the source code with the sub-computation related to `mean` highlighted. The x86 instructions corresponding to two of these statements are also shown. On the right, we show the model and the two types of masked modeling. The original input sequence is shown at the bottom with the edges denoting connectivity (derived from dependences). The row above shows the perturbed sequence and the perturbed edges. In particular, a positive edge (in blue) is selected and a negative edge (in red) is introduced. On top of the encoder, the output embeddings could be used to correctly classify the positive/negative edges.

4 EXPERIMENT

The implementation and pre-training details of CodeArt can be found in Section A of an extended version of this paper [70]. We evaluate CodeArt on three downstream tasks to demonstrate the effectiveness of the pre-training, including binary similarity, malware family classification, and type inference for binary executables. We aim to answer the following research questions.

- RQ1: How does CodeArt perform on binary similarity analysis?
- RQ2: How does CodeArt perform on malware family classification?
- RQ3: How does CodeArt perform on binary type inference?
- RQ4: Model analysis and ablation study of CodeArt.
- RQ5: How effective is CodeArt compared to GraphCodeBERT-like pre-training?

4.1 RQ1: Performance on Binary Similarity Analysis

Setup. Given an input binary function, a binary similarity analysis queries it among a pool of candidate functions, and tries to identify the function that is compiled from the same source code as the query function [75]. It plays a critical role in many security related tasks, such as one day vulnerability detection [24, 76], automatic software patching [66], and software plagiarism detection [49]. Machine-learning based binary similarity tools typically encode the query function and all the candidate functions to their embeddings. After that, the cosine similarity between the embeddings of the query function and each candidate function is computed. The candidate functions are then ranked by the similarity values. The function with the largest value is considered similar to the query function.

Dataset. We use the BinaryCorp-3M dataset [75] to finetune CodeArt on the binary similarity task. It is the same training dataset used by SOTA transformer-based models [75, 82, 87]. We use 7 real-world projects (i.e., Curl, Coreutils, Binutils, ImageMagick, SQLite, OpenSSL and Putty) as the test dataset. They are commonly used by the previous binary similarity works [53, 57, 76, 83].

Baseline. We compare the performance of CodeArt with the SOTA GNN-based model [53], *Graph Matching Network* (GMN)-based model [44], and two SOTA Transformer-based models Jtrans [75] and DiEmph [82]. GMN is a GNN-based model that takes as input a pair of programs and outputs a similarity value. It is worth noting that GMN does not generate embeddings for individual functions,

and can only be used for pair-wise similarity analysis. Note that although there are recent proposals that can achieve very good performance in binary similarity using advanced dynamic analysis such as [76, 83], these techniques require executing the functions (using seed inputs). They are hence not directly comparable.

Training Details. For both CodeArt and JTrans, we finetune their pre-trained models on the BinaryCorp-3M dataset [75]. Specifically, each training data sample is a triplet consisting of two binary functions compiled from a same source code function, and another binary function compiled from a different source code function. The training loss is the triplet loss enforcing a large cosine similarity between similar function pairs and a small similarity between dissimilar ones. For DiEmph, the GNN-based model, and the GMN-based model, we follow the scripts provided by the authors [53, 82] to train the models. For all models, we use the Coreutils project as the validation dataset to select best checkpoints, and report the performance on the remaining 6 projects.

Metrics. Following previous work [75], we use recall@1 as the metric to evaluate a binary similarity model. Specifically, suppose that we make N queries, recall@1 is computed as N divided by the number of queries that the function compiled from the same source code is correctly returned as the most similar function. We also adapt the setup of previous work to evaluate a binary similarity model with different sizes of candidate function pools. Intuitively, a larger pool size means a more challenging setup for a model.

Results. The results are shown in Fig. 8. We can see that CodeArt outperforms the GNN-based model and the GMN-based model by a large margin in all setups. The improvement is largely due to Transformer models' better capability of capturing long-range dependences in a data-rich scenario, compared to GNNs. For most projects, CodeArt significantly outperforms the Transformer-based models in challenging setups (i.e., a pool size larger than 100). The improvement demonstrates that CodeArt is able to encode program semantics more precisely. For Putty, CodeArt achieves comparable performance to the previous SOTA model DiEmph. We investigate the cases and found functions in Putty are shorter compared to other projects. For example, more than 75% functions in Putty have less than 50 instructions, while, on the other hand, more than half of the functions in Binutils are longer than that. For those relatively simple functions, the problem of spurious correlations (caused by the lack of symbols) in the baseline models is not as severe and thus the improvement introduced by CodeArt is not significant. Also, in simpler setups (i.e., a pool size smaller than 100), both DiEmph and CodeArt can achieve good performance.

To conclude, the results on the binary similarity task indicate CodeArt can generate embeddings that encode program semantics more precisely, benefiting more realistic use scenarios[75].

Zero-Shot Performance. The performance on the binary similarity task relies heavily on the quality of function embeddings generated by a pre-trained model. We thus further evaluate CodeArt on the binary similarity task with the zero-shot setup to measure the effectiveness of pre-training. Specifically, we directly use the embeddings generated by the pre-trained CodeArt model without finetuning it on the binary similarity task. We compare its performance to the pre-trained JTrans model, which is pre-trained on the same dataset as CodeArt. The results are shown in Table 1. We can see that CodeArt demonstrates strong zero-shot performance by achieving over 30% higher performance on all setups. It indicates that the function-level semantics are learned by CodeArt during the pre-training process. We also conduct t-test between CodeArt and JTrans' performance and results show that the improvement is statistically significant with p-values 1e-6, 6e-8, 1e-6, 2e-9, 3e-9 for pool size 32, 50, 100, 200, 500 respectively.

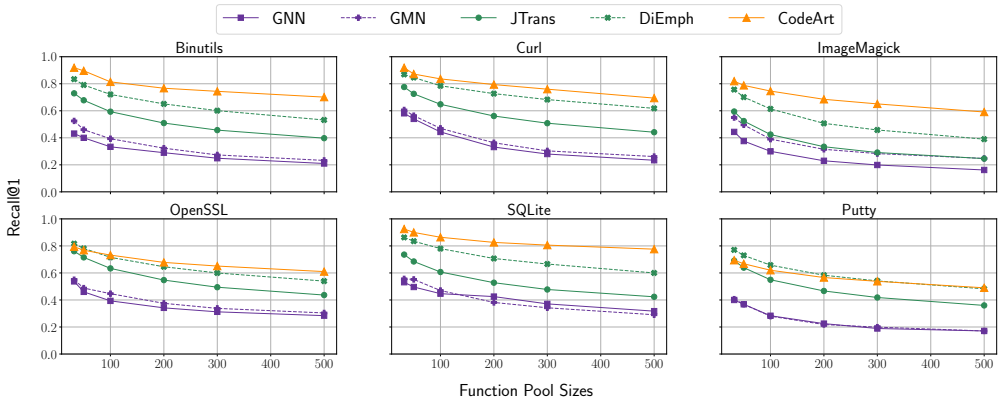


Fig. 8. Performance (recall@1) of CodeArt compared to the SOTA GNN-based model, GMN-based model and Transformer-based models. The x-axes denote different sizes of candidate function pools, and the y-axes denote the performance in terms of recall@1.

Table 1. Zero-Shot performance (recall@1) of CodeArt compared to JTrans. Each row lists performance of both models w.r.t. one binary project with different sizes of candidate function pools.

Programs	Pool size 32		Pool size 50		Pool size 100		Pool size 200		Pool size 500	
	JTrans	CodeArt	JTrans	CodeArt	JTrans	CodeArt	JTrans	CodeArt	JTrans	CodeArt
Binutils	0.30	0.71	0.26	0.68	0.20	0.60	0.14	0.57	0.09	0.48
Curl	0.32	0.71	0.28	0.66	0.21	0.63	0.16	0.56	0.12	0.47
ImageMagick	0.28	0.60	0.23	0.55	0.18	0.49	0.13	0.45	0.08	0.38
OpenSSL	0.33	0.61	0.28	0.55	0.22	0.52	0.17	0.49	0.12	0.43
SQLite	0.25	0.75	0.21	0.76	0.15	0.69	0.11	0.62	0.07	0.55
Putty	0.30	0.49	0.25	0.45	0.19	0.41	0.14	0.36	0.09	0.31
Average	**0.30**	**0.64**	**0.25**	**0.61**	**0.19**	**0.56**	**0.14**	**0.51**	**0.10**	**0.44**

4.2 RQ2: Performance on Malware Family Classification

Setup. To bypass malware detectors, the authors of a malware may create a set of variants of the malware by changing the code while keeping the malicious logic. Such a set of variants is often referred to as a malware family. Malware family classification aims to attribute a given malware sample to a known family. It can help many downstream security applications such as malware detection [81] and malware authorship attribution [33].

In many cases, it is very hard, if not impossible, to precisely attribute a malware sample to a single family. In our dataset, a malware sample may have multiple labels. We therefore use a multi-label classification setting which performs independent binary classification on each class.

Dataset. We build a dataset with the malware samples and labels provided by VirusTotal [3] and VirusShare [4]. Since real-world malware families follow a long tail distribution, we use the most frequent 100 families to build our classification dataset. It consists of 5484 malware samples. We randomly split the dataset into train, validation, and test set with a ratio of 70%, 10%, and 20%.

Baseline. To demonstrate the effectiveness of CodeArt, we choose JTrans as a Transformer-based baseline because it is pre-trained on the same dataset [75] as CodeArt. Moreover, a mainstream of

[3]https://www.virustotal.com
[4]https://virusshare.com

Table 2. Malware Family Classification

		AUC (↑)	LRAP (↑)	LRL (↓)
2 Funcs.	JTrans	0.705	0.205	0.300
	CodeArt	0.925	**0.597**	0.086
3 Funcs.	JTrans	0.898	0.478	0.111
	CodeArt	0.924	**0.597**	0.086
4 Funcs.	JTrans	0.901	0.489	0.107
	CodeArt	**0.928**	**0.597**	**0.084**
	CNN	0.834	0.326	0.183

Table 3. Ablation Study Using Zero-Shot Binary Similarity with Pool-Size 100 on Coreutils

	Recall@1	Recall@3	Recall@5	MRR
CodeArt-3M	0.394	0.548	0.604	0.486
w/o local mask	0.030	0.096	0.138	0.082
w/o trans-closure	0.346	0.498	0.590	0.447
max-trans-closure 4	0.322	0.504	0.574	0.429
max-trans-closure 6	0.298	0.456	0.562	0.406
w/o rel-pos-bias	0.226	0.376	0.464	0.335

malware family classification leverages Convolutional Neural Network (CNN) to extract features from malware samples [52, 55]. Due to the lack of available code artifacts, we additionally re-implement a CNN SOTA [52] based on ResNet-50 and use it as another baseline.

Metrics. We use three metrics typically used in multi-label classification tasks to measure the performance, i.e., ROC-AUC score, Label Ranking Average Precision score (LRAP), and Label Ranking Loss (LRL) [72]. ROC-AUC score is the averaged area under the ROC curve, where ROC curve is the plot of the true positive rate (TPR) against the false positive rate (FPR), at various thresholds. LRAP and LRL rank the predicted probability of all possible types, and compute the scores based on the rank of the groundtruth type. Intuitively, a higher rank of the ground-truth type indicates the model has better capability of identifying the correct type of the input code. Please refer to Section B in the extended version of this paper [70].

Training Details. CodeArt is a function-level encoder. To encode a whole binary to an embedding, we encode the forefront k functions separately by our encoder and use an attention pooling strategy [50] to pool the resulting k embeddings into a single binary-level embedding. With the binary-level embedding, we stack a standard classification head to do the prediction. We similarly implement a baseline classifier for JTrans. For the CNN-based baseline, we follow [52] to encode a malware binary to an image. Given a malware binary, we encode the entire binary byte-by-byte to a grey-scale image, compress it to a size of 256×256, and convert it according to ResNet-50's input format. Then we train a ResNet-50 classifier on the images encoded from malware samples.

Results. We report the averaged results on 100 classes in Table 2. Columns 3–5 list the three metrics. Each row presents the performance of one model/setup. Rows 1–2, 3–4, 5–6 present the related models when encoding the forefront 2,3,4 functions, respectively. The last row presents the CNN result, which is on whole binaries. We can see that CodeArt achieves the best performance when encoding the forefront 4 functions. It is worth noting that CodeArt achieves good performance when encoding even only 2 functions, better than CNN and JTrans. The performance of JTrans degrades when the number of encoded functions decreases because encoding fewer functions requires a model to understand individual functions more precisely. The improvement of CodeArt compared to JTrans is significant with p-values 3e-4, 2e-4, 3e-3 for AUC-ROC, LRAP, LRL respectively. These results demonstrate that the functional semantics learned in the pre-training of CodeArt is more informative for malware family classification. Detailed (per-class) results can be found in Section B in the extended version of this paper [70].

4.3 RQ3: Performance on Type Inference

Setup. Binary programs do not have high-level type information as in the source code (e.g., pointer, struct, array). The binary type inference task aims to recover the high-level type information from assembly code. Specifically, given a location (e.g., a register) in the binary code, a type inference tool outputs the possible high-level type corresponding to the location. It helps a reverse-engineer

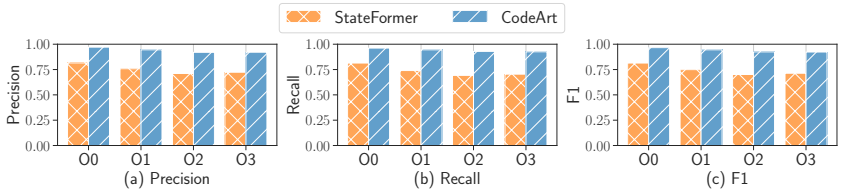


Fig. 9. Binary Type Inference Results on Different Optimization Levels

understand binary code. Moreover, the recovered type information is often an input for other analysis such as vulnerability detection [46], decompilation [5], and legacy code hardening[28]. Machine-learning based binary type inference tools typically formulate the type inference problem as a sequence labeling task [16, 56]. Following the setup of StateFormer [56], a SOTA binary type inference model, we define 35 common types as labels and include a special label `no-access` for tokens without groundtruth types.

Dataset. We reuse the dataset in StateFormer's repo [6]. As our model is only for x86-64, we only perform experiments on the x86-64 part although the original dataset includes multiple architectures. We randomly split the dataset into train, validation, and test sets with a ratio of 85%, 5%, and 10%.

Baseline. We directly compare to the reported results of StateFormer, which was finetuned on the same type inference dataset.

Training Details. We finetune CodeArt on the training set by stacking a 1-layer linear classification head to project the last-layer hidden states of tokens to class probabilities. We choose the best-performing checkpoint on the validation set and report the results on the test set.

Metrics. We follow the original setting of using precision, recall, and F1-score as the metrics. Here, precision $P = TP/(TP + FP)$, recall $R = TP/(TP + FN)$, and $F1 = 2 \times P \times R/(P + R)$, where TP is the number of correctly predicted types, FP is the number of incorrectly predicted types, and FN is the number of types incorrectly predicted as `no-access`.

Results on Different Optimization Levels. We show the results of CodeArt compared to StateFormer in Fig. 9. The three sub-figures show the performance of both models in terms of precision, recall, and F1, respectively. The x-axes denote the optimization levels. We can see that CodeArt achieves better performance in all optimization levels significantly with p-values 2e-8, 3e-8, and 3e-8 for precision, recall, and F1-score respectively. We attribute the improvement to CodeArt's capability of precisely encoding tokens in their dependence contexts. Moreover, the performance of CodeArt is more stable across different optimizations, suggesting CodeArt is more robust to syntactic code changes introduced by compiler optimizations.

4.4 RQ4: Model Analysis and Ablation Studies on What Helps CodeArt

Model Analysis. We now discuss CodeArt in detail. Architecture-wise, CodeArt largely aligns with the BERT-based models in hyperparameters and has slightly more trainable parameters (1296 in addition to 109 million) due to the additional parameters for relative positional bias. The average time of mask construction for a single sample is about 2.5ms. Note that mask construction can be done in parallel with the forwarding and back-propagation of the model. We leave the optimization to our future work. The pre-training dataset is the BinaryCorp-26M dataset [75]. We pre-train CodeArt on the dataset for 25k steps. For more pre-training details, please refer to Section A in the extended version of this paper [70].

[5] https://hex-rays.com/ida-pro/
[6] https://github.com/CUMLSec/stateformer/

Table 4. Ablation Study on Other Downstream Tasks that Require Fine-tuning

	MFC-AUC	MFC-LRAP	MFC-LRL	TI-PR	TI-Recall	TI-F1
CodeArt-3M	0.89	0.49	0.10	0.96	0.96	0.96
w/o local mask	0.86	0.40	0.13	0.92	0.91	0.91
w/o trans-closure	0.86	0.47	0.14	0.94	0.93	0.94
max-trans-closure 4	0.90	0.50	0.10	0.96	0.96	0.96
max-trans-closure 6	0.90	0.47	0.11	0.96	0.95	0.95
w/o rel-pos-bias	0.87	0.47	0.12	0.96	0.96	0.96

Ablation Studies. To study how each component in CodeArt contributes to the final performance, we alter the pre-training process with different setups and observe how the performance changes. As detailed in Section 4.1, the zero-shot performance on the binary similarity task reflects the quality of embeddings generated by a model and thus demonstrates the effectiveness of pre-training. Therefore, we leverage the zero-shot binary similarity performance on Coreutils (with a pool size of 100) as the metric for pre-training. Due to the limitation of computation resources, we conduct the ablation study on the BinaryCorp-3M dataset. Note that all other evaluations are conducted on the CodeArt model pre-trained on the larger BinaryCorp-26M dataset.

The results are in Table 3. Each row denotes a variant of CodeArt and the first row lists the performance achieved with the default setup (denoted by CodeArt-3M). We first remove the *local attention masks* from CodeArt (denoted by w/o local mask). Without the local masks, CodeArt degenerates to a pure masked language model, since all tokens can freely attend to each other and there is no mechanism to ensure that <INST> nodes are aligned with the corresponding instructions, rendering the dependence modeling ineffective. Hence, the performance dramatically drops.

To demonstrate the necessity of modeling transitive dependences in CodeArt, we implement a variant that does not include the *transitive dependence closure* when constructing the dependence attention masks (denoted by w/o trans-closure). That is, the variant mimics the behavior of neighborhood-local aggregating GNNs, meaning that an instruction can only attend to its direct neighbours in the dependence graph. It weakens the perception of transitive dependences. Observe that the performance degrades by 4.8%. It demonstrates that computing transitive closures and using connectivity graphs indeed helps.

We further limit the maximum dependence distances when computing the transitive closures. Specifically, max-trans-closure 4 and max-trans-closure 6 denote variants that only include nodes reachable within 4 and 6 dependence edges, respectively. We can see that the default CodeArt (which does not limit the maximum distance) surpasses the two variants.

Moreover, we remove the *relative positional bias* from CodeArt (denoted by w/o rel-pos-bias). That is, for a given instruction, all the instructions in its dependence context are considered having the same distance to it. We can see that the recall@1 degrades by 17%. It validates that the relative position bias indeed helps the model distinguish instructions with different distances, enabling the model to learn more precise semantics.

In addition to zero-shot binary similarity, we present the ablation study results for other tasks that require fine-tuning in Table 4, where "MFC" denotes "Malware Family Classification" and "TI" denotes "Type Inference". We can see that CodeArt's default setting demonstrates best overall performance, indicating its strong generalizability on all tasks.

4.5 RQ5: Comparison with GraphCodeBERT-like Pre-training

As discussed in Section 2, GraphCodeBERT (GCB) leverages data-flow information to enhance the pre-training process. We compare the pre-training of CodeArt to GCB's in terms of *optimization stability* and *zero-shot performance on the binary similarity task*.

Table 5. GrapCodeBERT-like Model Results on Zero-Shot Coreutils Binary Similarity with Pool-Size 100

	Recall@1	Recall@3	Recall@5	Recall@10	MRR
CodeArt-3M	0.394	0.548	0.604	0.696	0.486
GCB-like-default	0.030	0.092	0.152	0.264	0.085
GCB-like-weaker-reg	0.082	0.176	0.266	0.394	0.159

Training Details. Since the GCB pre-training code is not publicly available and the origin GCB only worked on source code, we implement a pre-training pipeline following GCB's design and use BinaryCorp-3M[75] to pre-train it for the binary code. We detail how we implement the loss of GCB on binary code in Section C in the extended version of this paper [70]. Specifically, we implement two variants of the training pipeline: (1) *GCB-like-default* that faithfully follows the loss design in GCB. (2) *GCB-like-weaker-reg* that makes the additional losses smaller which means weaker regularization for the MLM loss.

Optimization Stability. CodeArt has much better stability. The training curves of both CodeArt and GraphCodeBERT-like pre-training are shown in Fig. 11 in Section C in the extended version of this paper [70].

Performance Difference. As is shown in Table 5, *GCB-like-weaker-reg* has better performance than *GCB-like-default* due to its stabler optimization. However, both variants are not comparable to CodeArt, which demonstrates the effectiveness of our approach when symbols are lacking.

5 LIMITATIONS AND FUTURE WORK

The prototype of CodeArt is a pre-trained model for x86-64 binary programs. It can be used in different downstream applications such as binary similarity analysis and malware family classification. Currently, CodeArt is evaluated only on binary programs compiled with different optimizations in commonly-used compilers. The evaluation does not involve obfuscated code. The pre-training dataset is adapted from JTrans [75], which is compiled by GCC. Following the most challenging setup in existing work [82], our test dataset for the binary similarity task is compiled by Clang-O0 and GCC-O3. The type inference dataset is obtained from StateFormer [56], which is compiled by GCC. For the malware dataset, we do not know the precise compiler setups since they are real-world malware samples. While the idea of leveraging data dependences to regularize attentions can generalize to other CPU architectures and potentially to obfuscated code, it requires additional data collection and implementation (for the analysis part) efforts to study CodeArt 's performance on them. We leave it as our future work.

6 RELATED WORK

Language Models for Code. A large body of work focuses on pre-training transformer-based language models on source code [3, 20, 29, 40, 77]. They typically perceive source code as a sequence of tokens and utilize the rich natural language artifacts in software, such as symbol names and code comments, to help models understand code semantics. Due to the intrinsic structure of code, researchers have proposed to enhance pre-training with structural information, such as Abstract Syntax Trees (ASTs) [34]. A few works further explore ways to leverage program semantics to improve the quality of language models. For example, OSCAR [58] enhances an IR-level code model with operational semantics of programs. They augment model inputs with abstract program states obtained from static analysis. Their efforts are orthogonal to ours. GraphCodeBERT [35] enhances MLM with data-flow graph structure, and we provide a detailed comparison in Section 4.5. Unlike all the structure-enhanced approaches, our initiative is the first to understand the code structure

from a language perspective. Our encoding pipeline and pre-training methods explicitly model the code language characteristics.

Additionally, there are some transformer-based models on binary code designed for specific downstream tasks [39, 56, 57, 75]. In contrast, CodeArt is designed to pre-train a general model that supports various downstream tasks.

Neural Networks for Explicit Code Structure. Apart from transformer-based language models, numerous code models explicitly encode code structures for specific software engineering tasks, such as program differencing [31], aligning code across platforms [41], disassembling binary code [84], software maintenance [23], and code completion [71]. While these techniques leverage graph structures to derive embeddings, they usually require supervised training. Our empirical results demonstrate that integrating self-supervised learning and self-attention in Transformers with dependence graphs is not only feasible but can also lead to superior performance. Moreover, adapting these models for pre-training tasks requires non-trivial engineering efforts [6, 14, 42]. In contrast, CodeArt proposes a unique attention regularization method that is fully compatible with existing efficient and scalable implementations for NLP Transformers. Thus, the pre-training of CodeArt can easily scale to larger datasets.

Binary Program Analysis. CodeArt works at the level of assembly code, leveraging disassemblers [84] to decode binary files into textual assembly code. There exists a body of research focusing on binary program dependence analysis [85], which could enhance the analysis components in CodeArt. Our work is built upon these fundamental binary program analyses.

7 CONCLUSION

We introduce a novel method for pre-training Transformer-based code models in scenarios where symbols are lacking. This method features an innovative attention regularization technique that leverages program analysis to derive potential dependencies between instructions, subsequently forming attention masks. Our pre-trained model is general and can serve a wide range of applications. The empirical results show that our technique substantially outperforms the state-of-the-art, as well as GraphCodeBERT-like pre-trained models, in three downstream tasks, including binary similarity, malware family classification, and type inference for binary executables.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This research was supported, in part by DARPA VSPELLS - HR001120S0058, IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2024. CodeArt Artifact. https://doi.org/10.5281/zenodo.11096386

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[5] Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137.

[6] Uri Alon and Eran Yahav. 2020. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205* (2020).

[7] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 8626–8637.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.

[10] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 30–38.

[11] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).

[12] Yushi Cao, Zhiming Li, Tianpei Yang, Hao Zhang, Yan Zheng, Yi Li, Jianye Hao, and Yang Liu. 2022. GALOIS: boosting deep reinforcement learning via generalizable logic synthesis. *Advances in Neural Information Processing Systems* 35 (2022), 19930–19943.

[13] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

[14] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 3438–3445.

[15] Fuxiang Chen, Fatemeh H Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 401–412.

[16] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.

[17] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A unifying, graph-based view of program learning, understanding, and repair. *Advances in Neural Information Processing Systems* 34 (2021), 23089–23101.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

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

[20] Yangruibo Ding, Ben Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2023. TRACED: Execution-aware Pre-training for Source Code. *arXiv preprint arXiv:2306.07487* (2023).

[21] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Chen. 2023. Towards learning generalizable code embeddings using task-agnostic graph convolutional networks. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–43.

[22] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering* 27, 3 (2022), 63.

[23] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. FIRA: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International Conference on Software Engineering*. 970–981.

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

[25] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[26] Pengcheng Fang, Zhenhua Zou, Xusheng Xiao, and Zhuotao Liu. 2023. iSyn: Semi-automated Smart Contract Synthesis from Legal Financial Agreements. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 727–739.

[27] Sen Fang, Tao Zhang, Youshuai Tan, He Jiang, Xin Xia, and Xiaobing Sun. 2023. RepresentThemAll: A Universal Learning Representation of Bug Reports. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 602–614.

[28] Kassem Fawaz, Huan Feng, and Kang G Shin. 2015. Anatomization and protection of mobile apps' location privacy threats. In *24th USENIX Security Symposium (USENIX Security 15)*. 753–768.
[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.
[30] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[31] Lian Gao, Yu Qu, Sheng Yu, Yue Duan, and Heng Yin. [n. d.]. SIGMADIFF: Semantics-Aware Deep Graph Matching for Pseudocode Diffing. In *Network and Distributed System Security (NDSS) Symposium 2024*.
[32] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1933–1945.
[33] Jason Gray, Daniele Sgandurra, and Lorenzo Cavallaro. 2021. Identifying authorship style in malicious binaries: techniques, challenges & datasets. *arXiv preprint arXiv:2101.06124* (2021).
[34] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.
[35] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[36] Yuxiang Guo, Xiaopeng Gao, Zhenyu Zhang, WK Chan, and Bo Jiang. 2023. A study on the impact of pre-trained model on Just-In-Time defect prediction. *arXiv preprint arXiv:2309.02317* (2023).
[37] Yixin Guo, Pengcheng Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2020. GRAPHSPY: Fused Program Semantic-Level Embedding via Graph Neural Networks for Dead Store Detection. *arXiv preprint arXiv:2011.09501* (2020).
[38] IDA Pro 2023. *A powerful disassembler and a versatile debugger*. https://hex-rays.com/ida-pro/
[39] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. 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*. 1631–1645.
[40] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.
[41] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. 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*. 151–163.
[42] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.
[43] Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Soft-Labeled Contrastive Pre-training for Function-level Code Representation. *arXiv preprint arXiv:2210.09597* (2022).
[44] 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.
[45] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-Trained Model. *arXiv preprint arXiv:2305.10785* (2023).
[46] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. 1–1.
[47] 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. *arXiv preprint arXiv:1907.11692* (2019).
[48] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
[49] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. https://doi.org/10.1109/TSE.2017.2655046
[50] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[51] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. Graphcode2vec: Generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 524–536.

[52] Yixuan Ma, Shuang Liu, Jiajun Jiang, Guanhong Chen, and Keqiu Li. 2021. A comprehensive study on learning-based PE malware family classification methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1314–1325.

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

[54] Kollin Napier, Tanmay Bhowmik, and Shaowei Wang. 2023. An empirical study of text-based machine learning models for vulnerability detection. *Empirical Software Engineering* 28, 2 (2023), 38.

[55] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. 2011. Malware Images: Visualization and Automatic Classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security* (Pittsburgh, Pennsylvania, USA) *(VizSec '11)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2016904.2016908

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

[57] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

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

[59] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R Lyu. 2023. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. *arXiv preprint arXiv:2306.01394* (2023).

[60] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[61] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[62] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[63] Baptiste Rozière, 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).

[64] Iman Saberi and Fatemeh H Fard. 2023. Model-Agnostic Syntactical Information for Pre-Trained Programming Language Models. *arXiv preprint arXiv:2303.06233* (2023).

[65] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. 2023. SEAL: Integrating Program Analysis and Repository Mining. *ACM Transactions on Software Engineering and Methodology* (2023).

[66] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 6 (dec 2021), 36 pages. https://doi.org/10.1145/3412376

[67] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. *arXiv preprint arXiv:2304.05216* (2023).

[68] Yucen Shi, Ying Yin, Zhengkui Wang, David Lo, Tao Zhang, Xin Xia, Yuhai Zhao, and Bowen Xu. 2022. How to better utilize code graphs in semantic code search?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 722–733.

[69] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[70] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking. *arXiv preprint arXiv:2402.11842* (2024).

[71] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[72] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. 2010. Mining multi-label data. *Data mining and knowledge discovery handbook* (2010), 667–685.

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

[74] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*. 287–298.

[75] 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*. 1–13.

[76] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 319–330.

[77] 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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[78] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. *arXiv preprint arXiv:2305.18607* (2023).

[79] Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E Gonzalez, and Ion Stoica. 2021. Representing long-range context for graph neural networks with global attention. *Advances in Neural Information Processing Systems* 34 (2021), 13266–13279.

[80] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[81] Guangquan Xu, Meiqi Feng, Litao Jiao, Jian Liu, Hong-Ning Dai, Ding Wang, Emmanouil Panaousis, and Xi Zheng. 2021. MFF-AMD: multivariate feature fusion for Android malware detection. In *Collaborative Computing: Networking, Applications and Worksharing: 17th EAI International Conference, CollaborateCom 2021, Virtual Event, October 16-18, 2021, Proceedings, Part I 17*. Springer, 368–385.

[82] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1106–1118. https://doi.org/10.1145/3597926.3598121

[83] Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. *arXiv preprint arXiv:2308.15449* (2023).

[84] 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)*. 2709–2725.

[85] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31.

[86] Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. 2020. Beyond homophily in graph neural networks: Current limitations and effective designs. *Advances in neural information processing systems* 33 (2020), 7793–7804.

[87] 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 preprint arXiv:2308.12659* (2023).

Received 2023-09-29; accepted 2024-01-23