PalmTree: Learning an Assembly Language Model for Instruction Embedding

Xuezixiang Li
University of California Riverside
Riverside, CA 92521, USA
xli287@ucr.edu

Yu Qu
University of California Riverside
Riverside, CA 92521, USA
yuq@ucr.edu

Heng Yin
University of California Riverside
Riverside, CA 92521, USA
heng@cs.ucr.edu

ABSTRACT

Deep learning has demonstrated its strengths in numerous binary analysis tasks, including function boundary detection, binary code search, function prototype inference, value set analysis, etc. When applying deep learning to binary analysis tasks, we need to decide what input should be fed into the neural network model. More specifically, we need to answer how to represent an instruction in a fixed-length vector. The idea of automatically learning instruction representations is intriguing, but the existing schemes fail to capture the unique characteristics of disassembly. These schemes ignore the complex intra-instruction structures and mainly rely on control flow in which the contextual information is noisy and can be influenced by compiler optimizations.

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

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Theory of computation** → *Program analysis*; • **Computing methodologies** → *Knowledge representation and reasoning*.

KEYWORDS

Deep Learning, Binary Analysis, Language Model, Representation Learning

ACM Reference Format:

Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484587

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484587

1 INTRODUCTION

Recently, we have witnessed a surge of research efforts that leverage deep learning to tackle various binary analysis tasks, including function boundary identification [37], binary code similarity detection [23, 31, 40, 42, 43], function prototype inference [5], value set analysis [14], malware classification [35], etc. Deep learning has shown noticeably better performances over the traditional program analysis and machine learning methods.

When applying deep learning to these binary analysis tasks, the first design choice that should be made is: what kind of input should be fed into the neural network model? Generally speaking, there are three choices: we can either directly feed raw bytes into a neural network (e.g., the work by Shin et al. [37], αDiff [23], DeepVSA [14], and MalConv [35]), or feed manually-designed features (e.g., Gemini [40] and Instruction2Vec [41]), or automatically learn to generate a vector representation for each instruction using some representation learning models such as word2vec (e.g., InnerEye [43] and EKLAVYA [5]), and then feed the representations (embeddings) into the downstream models.

Compared to the first two choices, automatically learning instruction-level representation is more attractive for two reasons: (1) it avoids manually designing efforts, which require expert knowledge and may be tedious and error-prone; and (2) it can learn higher-level features rather than pure syntactic features and thus provide better support for downstream tasks. To learn instruction-level representations, researchers adopt algorithms (e.g., word2vec [28] and PV-DM [20]) from Natural Language Processing (NLP) domain, by treating binary assembly code as natural language documents.

Although recent progress in instruction representation learning (instruction embedding) is encouraging, there are still some unsolved problems which may greatly influence the quality of instruction embeddings and limit the quality of downstream models:

First, existing approaches ignore the complex internal formats of instructions. For instance, in x86 assembly code, the number of operands can vary from zero to three; an operand could be a CPU register, an expression for a memory location, an immediate constant, or a string symbol; some instructions even have implicit operands, etc. Existing approaches either ignore this structural information by treating an entire instruction as a word (e.g., InnerEye [43] and EKLAVYA [5]) or only consider a simple instruction format (e.g., Asm2Vec [10]). Second, existing approaches use Control Flow Graph (CFG) to capture contextual information between instructions (e.g., Asm2Vec [10], InnerEye [43], and the work by Yu et al. [42]). However, the contextual information on control flow can be noisy due to compiler optimizations, and cannot reflect the actual dependency relations between instructions.

Moreover, in recent years, pre-trained deep learning models [33] are increasingly attracting attentions in different fields such as Computer Vision (CV) and Natural Language Processing (NLP). The intuition of pre-training is that with the development of deep learning, the numbers of model parameters are increasing rapidly. A much larger dataset is needed to fully train model parameters and to prevent overfitting. Thus, pre-trained models (PTMs) using *large-scale unlabeled corpora* and *self-supervised training* tasks have become very popular in some fields such as NLP. Representative deep pre-trained language models in NLP include BERT [9], GPT [34], RoBERTa [24], ALBERT [19], etc. Considering the naturalness of programming languages [1, 16] including assembly language, it has great potential to pre-train an assembly language model for different binary analysis tasks.

To solve the existing problems in instruction representation learning and capture the underlying characteristics of instructions, in this paper, we propose a pre-trained assembly language model called PalmTree[1] for general-purpose instruction representation learning. PalmTree is based on the BERT [9] model but pre-trained with newly designed training tasks exploiting the inherent characteristics of assembly language.

We are not the first to utilize the BERT model in binary analysis. For instance, Yu et al. [42] proposed to take CFG as input and use BERT to pre-train the token embeddings and block embeddings for the purpose of binary code similarity detection. Trex [31] uses one of BERT's pre-training tasks – Masked Language Model (MLM) to learn program execution semantics from functions' micro-traces (a form of under-constrained dynamic traces) for binary code similarity detection.

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

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

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

In summary, we have made the following contributions:

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

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

2 BACKGROUND

In this section, we firstly summarize existing approaches and background knowledge of instruction embedding. Then we discuss some unsolved problems of the existing approaches. Based on the discussions, we summarize representative techniques in this field.

2.1 Existing Approaches

Based on the embedding generation process, existing approaches can be classified into three categories: raw-byte encoding, manually-designed encoding, and learning-based encoding.

2.1.1 Raw-byte Encoding. The most basic approach is to apply a simple encoding on the raw bytes of each instruction, and then feed the encoded instructions into a deep neural network. One such encoding is "one-hot encoding", which converts each byte into a 256-dimensional vector. One of these dimensions is 1 and the others are all 0. MalConv [35] and DeepVSA [14] take this approach to classify malware and perform coarse-grained value set analysis, respectively.

One instruction may be several bytes long. To strengthen the sense of an instruction, DeepVSA further concatenates the one-hot vectors of all the bytes belonging to an instruction, and forms a vector for that instruction.

Shin et al. [37] take a slightly different approach to detect function boundaries. Instead of a one-hot vector, they encode each byte as a 8-dimensional vector, in which each dimension presents a corresponding digit in the binary representation of that byte. For instance, the **`0x90`** will be encoded as

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

In general, this kind of approach is simple and efficient, because it does not require disassembly, which can be computationally

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

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

2.1.2 Manual Encoding of Disassembled Instructions. Knowing that disassembly carries more semantic information about an instruction, this approach first disassembles each instruction and encodes some features from the disassembly.

Li et al. [21] proposed a very simple method, which only extracts opcode to represent an instruction, and encodes each opcode as a one-hot vector. Unfortunately, this method completely ignores the information from operands. Instruction2Vec [41] makes use of both opcode and operand information. Registers, addresses, and offsets are encoded in different ways, and then concatenated to form a vector representation. Each instruction is encoded as a nine-dimensional feature vector. An instruction is divided into tokens, and tokens are encoded as unique index numbers. While an opcode takes one token, a memory operand takes up to four tokens, including base register, index register, scale, and displacement.

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

2.1.3 Learning-based Encoding. Inspired by representation learning in other domains such as NLP (e.g., word2vec [27, 28]), we would like to automatically learn a representation for each instruction that carries higher-level semantic information. Then this instruction-level representation can be used for any downstream binary analysis tasks, achieving high analysis accuracy and generality.

Several attempts have been made to leverage word2vec [28] to automatically learn instruction-level representations (or embeddings), for code similarity detection [26, 43] and function type inference [5], respectively. The basic idea of this approach is to treat each instruction as a word, and each function as a document. By applying a word2vec algorithm (Skip-gram or CBOW [27, 28]) on the disassembly code in this way, we can learn a continuous numeric vector for each instruction.

In order to detect similar functions in binary code, Asm2Vec [10] makes use of the PV-DM model [20] to generate instruction embeddings and an embedding for the function containing these instructions simultaneously. Unlike the above approach that treats each instruction as a word, Asm2Vec treats each instruction as one opcode and up to two operands and learns embeddings for opcodes and operands separately.

2.2 Challenges in Learning-based Encoding

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

2.2.1 Complex and Diverse Instruction Formats. Instructions (especially those in CISC architectures) are often in a variety of formats, with additional complexities. Listing 1 gives several examples of instructions in x86.

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

Listing 1: Instructions are complex and diverse

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

A good instruction-level representation must understand these internal details about each instruction. Unfortunately, the existing learning-based encoding schemes do not cope with these complexities very well. Word2vec, adopted by some previous efforts [5, 26, 43], treats an entire instruction as one single word, totally ignoring these internal details about each instruction.

Asm2Vec [10] looks into instructions to a very limited degree. It considers an instruction having one opcode and up to two operands. In other words, each instruction has up to three tokens, one for opcodes, and up to two for operands. A memory operand with an expression will be treated as one token, and thus it does not understand how a memory address is calculated. It does not take into account other complexities, such as prefix, a third operand, implicit operands, **EFLAGS**, etc.

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

Listing 2: Instructions can be reordered

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

Table 1: Summary of Approaches

Name	Encoding	Internal Structure	Context	Disassembly Required
DeepVSA [14]	1-hot encoding on raw-bytes	no	no	no
Instruction2Vec [41]	manually designed	yes	no	yes
InnerEye [43]	word2vec	no	control flow	yes
Asm2Vec [10]	PV-DM	partial	control flow	yes
PalmTree (this work)	BERT	yes	control flow & data flow	yes

While this assumption might hold in general, compiler optimizations tend to break this assumption to maximize instruction level parallelism. In particular, compiler optimizations (e.g., "-fschedule-insns", "-fmodulo-sched", "-fdelayed-branch" in GCC) seek to avoid stalls in the instruction execution pipeline by moving the load from a CPU register or a memory location further away from its last store, and inserting irrelevant instructions in between.

Listing 2 gives an example. The **`test`** instruction at Line 10 has no relation with its surrounding **`call`** and **`mov`** instructions. The **`test`** instruction, which will store its results into **`EFLAGS`**, is moved before the **`mov`** instruction by the compiler, such that it is further away from the **`je`** instruction at Line 14, which will use (load) the **`EFLAGS`** computed by the **`test`** instruction at Line 10. From this example, we can see that contextual relations on the control flow can be noisy due to compiler optimizations.

Note that instructions also depend on each other via data flow (e.g., lines 8 and 12 in Listing 2). Existing approaches only work on control flow and ignore this important information. On the other hand, it is worth noting that most existing PTMs cannot deal with the sequence longer than 512 tokens [33] (PTMs that can process longer sequences, such as Transformer XL [8], will require more GPU memory), as a result, even if we directly train these PTMs on instruction sequences with MLM, it is hard for them capture long range data dependencies which may happen among different basic blocks. Thus a new pre-training task capturing data flow dependency is desirable.

2.3 Summary of Existing Approaches

Table 1 summarizes and compares the existing approaches, with respect to which encoding scheme or algorithm is used, whether disassembly is required, whether instruction internal structure is considered, and what context is considered for learning. In summary, raw-byte encoding and manually-designed encoding approaches are too rigid and unable to convery higher-level semantic information about instructions, whereas the existing learning-based encoding approaches cannot address challenges in instruction internal structures and noisy control flow.

3 DESIGN OF PALMTREE

3.1 Overview

To meet the challenges summarized in Section 2, we propose PalmTree, a novel instruction embedding scheme that automatically learns a language model for assembly code. PalmTree is based on BERT [9], and incorporates the following important design considerations.

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

Then, in order to train the deep neural network to understand the internal structures of instructions, we make use of a recently proposed training task in NLP to train the model: Masked Language Model (MLM) [9]. This task trains a language model to predict the masked (missing) tokens within instructions.

Moreover, we would like to train this language model to capture the relationships between instructions. To do so, we design a training task, inspired by word2vec [28] and Asm2Vec [10], which attempts to infer the word/instruction semantics by predicting two instructions' co-occurrence within a sliding window in control flow. We call this training task Context Window Prediction (CWP), which is based on Next Sentence Prediction (NSP) [9] in BERT. Essentially, if two instructions i and j fall within a sliding window in control flow and i appears before j, we say i and j have a contextual relation. Note that this relation is more relaxed than NSP, where two sentences have to be next to each other. We make this design decision based on our observation described in Section 2.2.2: instructions may be reordered by compiler optimizations, so adjacent instructions might not be semantically related.

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

Figure 1 presents the design of PalmTree. It consists of three components: Instruction Pair Sampling, Tokenization, and Language Model Training. The main component (Assembly Language Model) of the system is based on the BERT model [9]. After the training process, we use mean pooling of the hidden states of the second last layer of the BERT model as instruction embedding. The Instruction Pair Sampling component is responsible for sampling instruction pairs from binaries based on control flow and def-use relations.

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

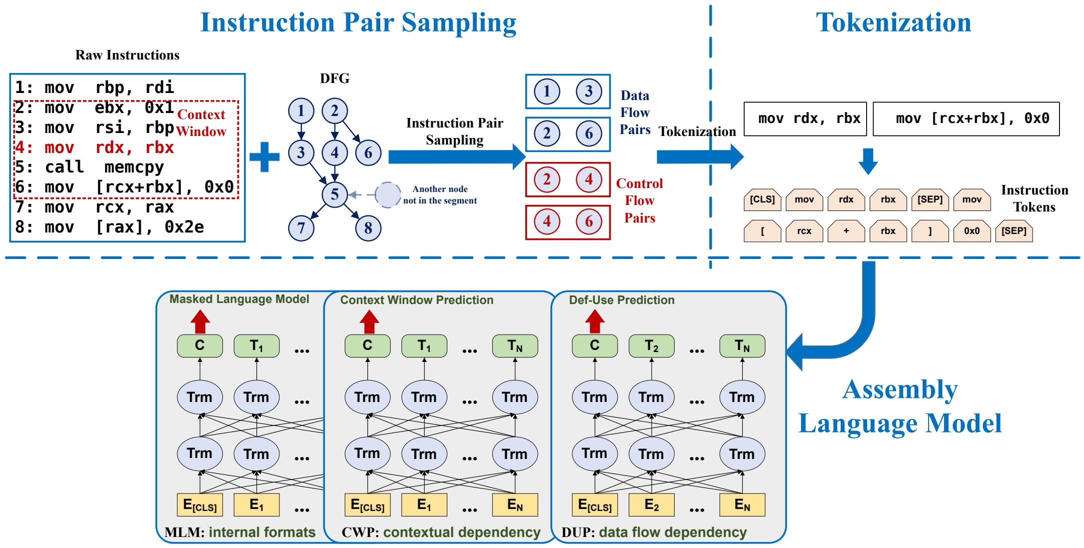

Figure 1: System design of PalmTree. `Trm` is the transformer encoder unit, `C` is the hidden state of the first token of the sequence (classification token), T_n ($n = 1 \ldots N$) are hidden states of other tokens of the sequence

embedding generation. In general, the tokenization strategy and MLM will help us address the first challenge in Section 2.2, and CWP and DUP can help us address the second challenge.

In Section 3.2, we introduce how we construct two kinds of instruction pairs. In Section 3.3, we introduce our tokenization process. Then, we introduce how we design different training tasks to pre-train a comprehensive assembly language model for instruction embedding in Section 3.4.

3.2 Input Generation

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

3.3 Tokenization

As introduced earlier, unlike Asm2Vec [10] which splits an instruction into opcode and up to two operands, we apply a more fine-grained strategy. For instance, given an instruction "`mov rax, qword [rsp+0x58]`", we divide it into "`mov`", "`rax`", "`qword`", "`[`", "`rsp`", "`+`", "`0x58`", and "`]`". In other words, we consider each instruction as a sentence and decompose the operands into more basic elements.

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

3.4 Assembly Language Model

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

3.4.1 PalmTree model. Our model is based on BERT [9], the state-of-the-art PTM in many NLP tasks. The proposed model is a multi-layer bidirectional transformer encoder. Transformer, firstly introduced in 2017 [39], is a neural network architecture solely based on multi-head self attention mechanism. In PalmTree, transformer units are connected bidirectionally and stacked into multiple layers.

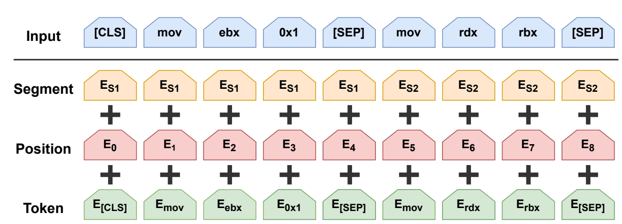

Figure 2: Input Representation

We treat each instruction as a sentence and each token as a word. Instructions from control flow and data flow sequences are

[2] https://binary.ninja/

concatenated and then fed into the BERT model. As shown in Figure 2, the first token of this concatenated input is a special token – **[CLS]**, which is used to identify the start of a sequence. Secondly, we use another token **[SEP]** to separate concatenated instructions. Furthermore, we add position embedding and segment embedding to token embedding, and use this mixed vector as the input of the bi-directional transformer network, as shown in Figure 2. Position embedding represents different positions in the input sequence, while segment embedding distinguishes the first and second instructions. Position embedding and segment embedding will be trained along with token embeddings. These two embeddings can help dynamically adjust token embeddings according to their locations.

3.4.2 Training task 1: Masked Language Model. The first task we use to pre-train PalmTree is Masked Language Model (MLM), which was firstly introduced in BERT [9]. Here is an example shown in Figure 3. Assuming that t_i denotes a token and instruction $I = t_1, t_2, t_3, ..., t_n$ consists of a sequence of tokens. For a given input instruction I, we first randomly select 15% of the tokens to replace. For the chosen tokens, 80% are masked by **[MASK]** (mask-out tokens), 10% are replaced with another token in the vocabulary (corrupted tokens), and 10% of the chosen tokens are unchanged. Then, the transformer encoder learns to predict the masked-out and corrupted tokens, and outputs a probability for predicting a particular token $t_i = [MASK]$ with a softmax layer located on the top of the transformer network:

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

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

$$\mathcal{L}_{MLM} = -\sum_{t_i \in m(I)} \log p(\hat{t}|I) \tag{2}$$

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

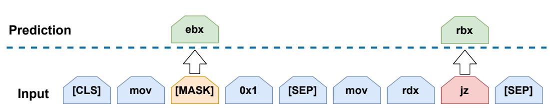

Figure 3: Masked Language Model (MLM)

Figure 3 shows an example. Given an instruction pair "**mov ebx, 0x1; mov rdx, rbx**", we first add special tokens **[CLS]** and **[SEP]**. Then we randomly select some tokens for replacement. Here we select **ebx** and **rbx**. The token **ebx** is replaced by the **[MASK]** token (the yellow box). The token **rbx** is replaced by the token **jz** (another token in the vocabulary, the red box). Next, we feed this modified instruction pair into the PalmTree model. The model will make a prediction for each token. Here we care about the predictions of the yellow and red boxes, which are the green boxes in Figure 3. Only the predictions of those two special tokens are considered in calculating the loss function.

3.4.3 Training task 2: Context Window Prediction. We use this training task to capture control flow information. Many downstream tasks [5, 14, 40, 43] rely on the understanding of contextual relations of code sequences in functions or basic blocks. Instead of predicting the whole following sentence (instruction) [18, 38], we perform a binary classification to predict whether the two given instructions co-occur within a context window or not, which makes it a much easier task compared to the whole sentence prediction. However, unlike natural language, control flows do not have strict dependencies and ordering. As a result, strict Next Sentence Prediction (NSP), firstly proposed by BERT [9], may not be suitable for capturing contextual information of control flow. To tackle this issue, we extend the context window, i.e., we treat each instruction w steps before and w steps after the target instruction in the same basic block as contextually related. w is the context windows size. In Section C.3, we evaluate the performance of different context window sizes, and pick $w = 2$ accordingly. Given an instruction I and a candidate instruction I_{cand} as input, the candidate instruction can be located in the contextual window of I, or a negative sample randomly selected from the dataset. $\hat{y}$ denotes the prediction of this model. The probability that the candidate instruction I_{cand} is a context instruction of I is defined as

$$p(\hat{y}|I, I_{cand}) = \frac{1}{1 + exp(\Theta(I \parallel I_{cand})_{cls})} \tag{3}$$

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

$$\mathcal{L}_{CWP} = -\sum_{I \in \mathcal{D}} \log p(\hat{y}|I, I_{cand}) \tag{4}$$

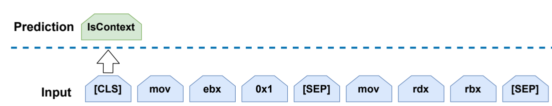

Figure 4: Context Window Prediction (CWP)

Here is an example in Figure 4. We use the input mentioned above. We feed the unchanged instruction pairs into the PalmTree model and pick the first output vector. We use this vector to predict whether the input are located in the same context window or not. In this case, the two instructions are next to each other. Therefore the correct prediction would be "true".

3.4.4 Training task 3: Def-Use Prediction. To further improve the quality of our instruction embedding, we need not only control flow information but also data dependency information across instructions. Sentence Ordering Prediction (SOP), first introduced by Lan et al. [19], is a very suitable choice. This task can help the PalmTree model to understand the data relation through DFGs, and we call it Def-Use Prediction (DUP).

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

$$p(\hat{y}|I_1, I_2) = \frac{1}{1 + exp(\Theta(I_1 \parallel I_2)_{cls})} \quad (5)$$

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

$$\mathcal{L}_{DUP} = -\sum_{I \in \mathcal{D}} p(\hat{y}|I_1, I_2) \quad (6)$$

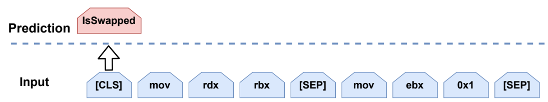

Figure 5: Def-Use Prediction (DUP)

We show an example in Figure 5. We still use the instruction pair discussed in Figure 4, but here we swap the two instructions. So the sequence is "`[CLS] mov rdx rbx [SEP] mov ebx 0x1 [SEP]`". We feed it into PalmTree and use the first output vector to predict whether this instruction pair remains unswapped or not. In this case, it should be predicted as "false" (which means this pair is swapped).

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

$$\mathcal{L} = \mathcal{L}_{MLM} + \mathcal{L}_{CWP} + \mathcal{L}_{DUP} \quad (7)$$

3.4.5 Instruction Representation. The transformer encoder produces a sequence of hidden states as output. There are multiple ways to generate instruction embeddings from the output. For instance, applying a max/mean pooling. We use mean pooling of the hidden states of the second last layer to represent the whole instruction. This design choice has the following considerations. First, the transformer encoder encodes all the input information into the hidden states. A pooling layer is a good way to utilize the information encoded by transformer. Second, results in BERT [9] also suggest that hidden states of previous layers before the last layer have offer more generalizability than the last layer for some downstream tasks. We evaluated different layer configurations and reported the results in Section C.2.

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

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

In the second way (fine-tuning), PalmTree is fine-tuned and trained together with the downstream model. This scheme will usually provide extra benefits when enough computing resources and training budget are available. There are several fine-tuning strategies [33], e.g., two-stage fine-tuning, multi-task fine-tuning.

4 EVALUATION

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

4.1 Evaluation Methodology

Intrinsic Evaluation. In NLP domain, intrinsic evaluation refers to the evaluations that compare the generated embeddings with human assessments [2]. Hence, for each intrinsic metric, manually organized datasets are needed. This kind of dataset could be collected either in laboratory on a limited number of examinees or through crowd-sourcing [25] by using web platforms or offline survey [2]. Unlike the evaluations in NLP domain, programming languages including assembly language (instructions) do not necessarily rely on human assessments. Instead, each opcode and operand in instructions has clear semantic meanings, which can be extracted from instruction reference manuals. Furthermore, debug information generated by different compilers and compiler options can also indicate whether two pieces of code are semantically equivalent. More specifically, we design two intrinsic evaluations: *instruction outlier detection* based on the knowledge of semantic meanings of opcodes and operands from instruction manuals, and *basic block search* by leveraging the debug information associated with source code.

Extrinsic Evaluation. Extrinsic evaluation aims to evaluate the quality of an embedding scheme along with a downstream machine learning model in an end-to-end manner [2]. So if a downstream model is more accurate when integrated with instruction embedding scheme A than the one with scheme B, then A is considered better than B. In this paper, we choose three different binary analysis tasks for extrinsic evaluation, i.e., Gemini [40] for *binary code similarity detection*, EKLAVYA [5] for *function type signatures inference*, and DeepVSA [14] for *value set analysis*. We obtained the original implementations of these downstream tasks for this evaluation. All of the downstream applications are implemented based on TensorFlow[3]. Therefore we choose the first way of deploying PalmTree in extrinsic evaluations (see Section 3.4.6). We encoded all

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

the instructions in the corresponding training and testing datasets and then fed the embeddings into downstream applications.

4.2 Experimental Setup

Baseline Schemes and PalmTree Configurations. We choose Instruction2Vec, word2vec, and Asm2Vec as baseline schemes. For fair comparison, we set the embedding dimension as 128 for each model. We performed the same normalization method as PalmTree on word2vec and Asm2Vec. We did not set any limitation on the vocabulary size of Asm2Vec and word2vec. We implemented these baseline embedding models and PalmTree using PyTorch [30]. PalmTree is based on BERT but has fewer parameters. While in BERT $\#Layers = 12$, $Head = 12$ and $Hidden\_dimension = 768$, we set $\#Layers = 12$, $Head = 8$, $Hidden\_dimension = 128$ in PalmTree, for the sake of efficiency and training costs. The ratio between the positive and negative pairs in both CWP and DUP is 1:1.

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

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

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

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

4.3 Intrinsic Evaluation

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

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

We classify instructions into 12 categories based on their opcode, according to the x86 Assembly Language Reference Manual [29]. More details about this process can be found in Table 8 in the Appendix. We prepared 50,000 instruction sets. Each set consists of four instructions from the same opcode category and one instruction from a different category.

Table 2: Intrinsic Evaluation Results, Avg. denotes the average of accuracy scores, and Stdev. denotes the standard deviation

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

Similarly, we classify instructions based on their operands. Table 9 in the Appendix provides details about this process. Essentially, we classify operand lists, according to the number of operands as well as the operand types. We created another 50,000 sets of instructions covering 10 categories, and each set contains four instructions coming from the same category, and one from a different category.

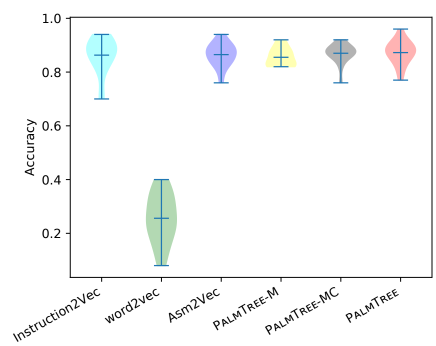

Figure 6: Accuracy of Opcode Outlier Detection

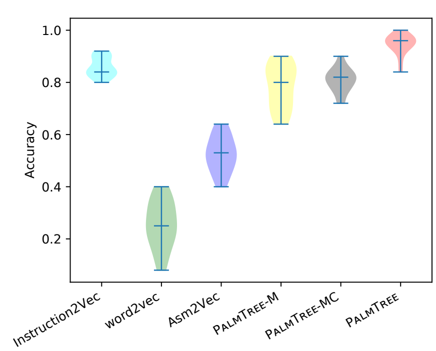

Figure 7: Accuracy of Operands Outlier Detection

The first and second columns of Table 2 present the accuracy distributions for opcode outlier detection and operand outlier detection respectively. We can make the following observations: (1) word2vec performs poorly in both experiments, because it does not take into account the instruction internal structures; (2) Instruction2Vec, as a manually-designed embedding, performs generally well in both experiments, because this manual design indeed takes different opcodes and operands into consideration; (3) Asm2Vec performs slightly better than Instruction2Vec in opcode outlier detection, but considerably worse in operand outlier detection, because its modeling for operands is not fine-grained enough; (4) Even though PalmTree-M and PalmTree-MC do not show obvious advantages over Asm2Vec and Instruction2Vec, PalmTree has the best accuracy in both experiments, which demonstrate that this automatically learned representation can sufficiently capture semantic differences in both opcodes and operands; and (5) All the three pre-training tasks contribute positively to PalmTree in both outlier detection experiments. Particularly, the DUP training task considerably boots the accuracy in both experiments, demonstrating that the def-use relations between instructions indeed help learn the assembly language model. A complete result of outlier detection can be found in Figure 6 and Figure 7.

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

We use `openssl-1.1.0h` and `glibc-2.29.1` as the testing set, which is not included in our training set. We compile them with O1, O2, and O3 optimization levels. We use the same method used in DeepBinDiff [11], which relies on the debug information from the program source code as the ground truth.

Figure 8 shows the ROC curves of Instruction2Vec, word2vec, Asm2Vec, and PalmTree for basic block search. Table 2 further lists the AUC (Area Under the Curve) score for each embedding scheme. We can observe that (1) word2vec, once again, has the worst performance; (2) the manually-designed embedding scheme, Instruction2Vec, is even better than word2vec, an automatically learned embedding scheme; (3) Asm2Vec performs reasonably well, but still worse than three configurations of PalmTree; and (4) The three PalmTree configurations have better AUC than other baselines, while consecutive performance improvements are observed.

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

4.4 Extrinsic Evaluation

An extrinsic evaluation reflects the ability of an instruction embedding model to be used as an input of downstream machine learning algorithms for one or several specific tasks [2]. As introduced earlier, we select three downstream tasks in binary analysis field, which are

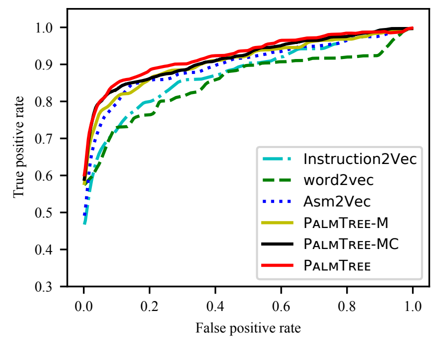

Figure 8: ROC curves for Basic Block Search

binary code similarity detection, function type signature analysis, and value set analysis.

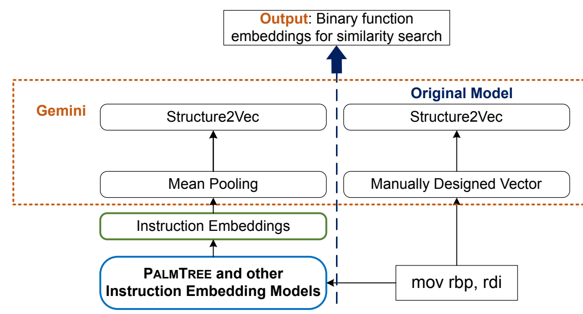

Figure 9: Instruction embedding models and the downstream model Gemini

4.4.1 Binary Code Similarity Detection. Gemini [40] is a neural network-based approach for cross-platform binary code similarity detection. The model is based on Structure2Vec [7] and takes ACFG (Attributed Control Flow Graph) as input. In an ACFG, each node is a manually formed feature vector for each basic block. Table 3 shows the attributes (i.e., features) of a basic block in the original implementation.

Table 3: Attributes of Basic Blocks in Gemini [40]

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

In this experiment, we evaluate the performance of Gemini, when having Instruction2Vec, word2vec, Asm2Vec, PalmTree-M, PalmTree-MC, and PalmTree as input, respectively. Moreover, we also used one-hot vectors with an embedding layer as a kind of instruction embedding (denoted as "one-hot") as another baseline.

The embedding layer will be trained along with Gemini. Figure 9 shows how we adopt different instruction embedding models to Gemini. Since Gemini takes a feature vector for each basic block, we use mean pooling to generate basic block embeddings based on embeddings of the instructions in the corresponding basic block. The architectures of our modified model and the original model are both shown in Figure 9. We also included its original basic block features as an additional baseline (denoted as "Gemini") for comparison.

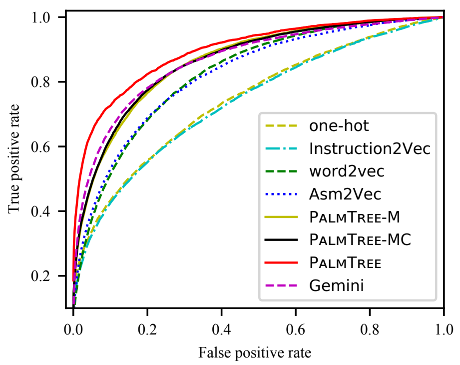

Figure 10: ROC curves of Gemini

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

Table 4: AUC values of Gemini

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

Table 4 gives the AUC values of Gemini when different models are used to generate its input. Figure 10 shows the ROC curves of Gemini when different instruction embedding models are used. Based on Table 4, we can make the following observations:

(1) Although the original paper [40] reported very encouraging performance of Gemini, we can observe that the original Gemini model does not generalize very well to completely new testing data.
(2) The manually designed embedding schemes, Instruction2Vec and one-hot vector, perform poorly, signifying that manually selected features might be only suitable for specific tasks.
(3) Despite that the testing set is considerably different from the training set, PalmTree can still perform reasonably well and beat the remaining schemes, demonstrating that PalmTree can substantially boost the generalizability of downstream tasks.
(4) All the three pre-training tasks contribute to the final model (PalmTree) for Gemini. However, both PalmTree-M and PalmTree-MC do not show obvious advantages over other baselines, signifying that only the complete PalmTree with the three training tasks can generate better embeddings than previous approaches in this downstream task.

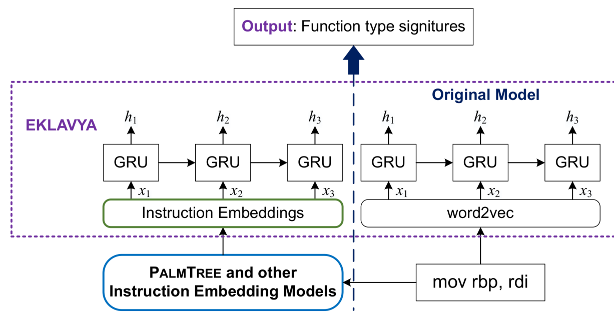

Figure 11: Instruction embedding models and EKLAVYA

4.4.2 Function Type Signature Inference. Function type signature inference is a task of inferring the number and primitive types of the arguments of a function. To evaluate the quality of instruction embeddings in this task, we select EKLAVYA, an approach proposed by Chua et al. [5]. It is based on a multi-layer GRU (Gated Recurrent Unit) network and uses word2vec as the instruction embedding method. According to the original paper, word2vec was pre-trained with the whole training dataset. Then, they trained a GRU network to infer function type signatures.

In this evaluation, we test the performances of different types of embeddings using EKLAVYA as the downstream application. Since the original model is not an end-to-end model, we do not need an embedding layer between instruction embeddings and the GRU network. We replaced the original word2vec in EKLAVYA with one-hot encoding, Instruction2Vec, Asm2Vec, PalmTree-M, PalmTree-MC, and PalmTree, as shown in Figure 11. Similarly, in order to evaluate the generalizability of the trained downstream models, we used very different training and testing sets (the same datasets described in Section 4.4.1).

Table 5: Accuracy and Standard Deviation of EKLAVYA

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

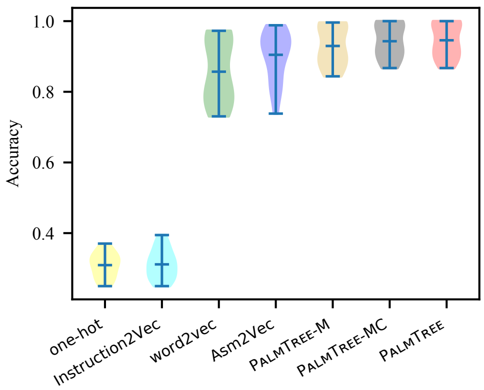

Figure 12: Accuracy of EKLAVYA

Table 5 and Figure 12 presents the accuracy of EKLAVYA on the testing dataset. Figure 15, and Figure 16 in the Appendix shows the loss value and accuracy of EKLAVYA during training and testing. From the results we can make the following observations:

(1) PalmTree and Asm2Vec can achieve higher accuracy than word2vec, which is the original choice of EKLAVYA.
(2) PalmTree has the best accuracy on the testing dataset, demonstrating that EKLAVYA when fed with PalmTree as instruction embeddings can achieve the best generalizability. Moreover, CWP contributes more (see PalmTree-MC), which implies that control-flow information plays a more significant role in EKLAVYA.
(3) Instruction2Vec performs very poorly in this evaluation, signifying that, when not done correctly, manual feature selection may disturb and mislead a downstream model.
(4) The poor results of one-hot encoding show that a good instruction embedding model is indeed necessary. At least in this task, it is very difficult for the deep neural network to learn instruction semantic through end-to-end training.

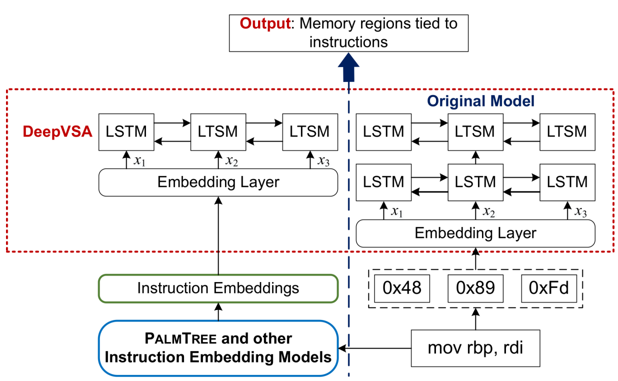

Figure 13: Instruction embedding models and the downstream model DeepVSA

4.4.3 Value Set Analysis. DeepVSA [14] makes use of a hierarchical LSTM network to conduct a coarse-grained value set analysis, which characterizes memory references into regions like global, heap, stack, and other. It feeds instruction raw bytes as input into a multi-layer LSTM network to generate instruction embeddings. It then feeds the generated instruction representations into another multi-layer bi-directional LSTM network, which is supposed to capture the dependency between instructions and eventually predict the memory access regions.

In our experiment, we use different kinds of instruction embeddings to replace the original instruction embedding generation model in DeepVSA. We use the original training and testing datasets of DeepVSA and compare prediction accuracy of different kinds of embeddings. The original datasets contain raw bytes only, thus we need to disassemble these raw bytes. After that we tokenize and encode these disassembled instructions for training and testing. We add an embedding layer before the LSTM network to further adjust instruction embeddings, as shown in Figure 13.

We use part of the dataset provided by the authors of DeepVSA. The whole dataset provided by the authors has 13.8 million instructions for training and 10.1 million for testing. Our dataset has 9.6 million instructions for training and 4.8 million for testing, due to the disassembly time costs. As explained in their paper [14], their dataset also used Clang and GCC as compilers and had no overlapping instructions between the training and testing datasets.

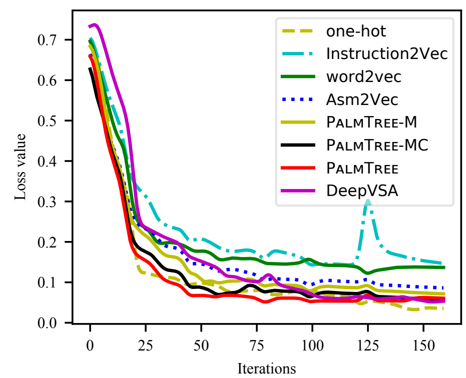

Figure 14: Loss value of DeepVSA during training

Table 6 lists the experimental results. We use Precision (P), Recall (R), and F1 scores to measure the performance. Figure 14 depicts the loss values of DeepVSA during training, when different instruction embedding schemes are used as its input. From these results, we have the following observations:

(1) PalmTree has visibly better results than the original DeepVSA and the other baselines in Global and Heap, and has slightly better results in Stack and Other since other baselines also have scores greater than 0.9.
(2) The three training tasks of PalmTree indeed contribute to the final result. It indicates that PalmTree indeed captures the data flows between instructions. In comparison, the other instruction embedding models are unable to capture data dependency information very well.
(3) PalmTree converged faster than original DeepVSA (see Figure 14), indicating that instruction embedding model can accelerate the training phase of downstream tasks.

Table 6: Results of DeepVSA

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

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

4.5 Runtime Efficiency

In this section, we conduct an experiment to evaluate runtime efficiencies of PalmTree and baseline approaches. First, we test the runtime efficiencies of different instruction embedding approaches. Second, we test the runtime efficiency of PalmTree when having different embedding sizes. We use 64, 128, 256, and 512 as embedding sizes, while 128 is the default setting. In the transformer encoder of PalmTree, the width of each feed-forward hidden layer is fixed and related to the size of the final output layer, which is 4 times of the embedding size [19]. We use `Coreutils-8.30` as the dataset. It includes 107 binaries and 1,006,169 instructions. We disassembled the binaries with Binary Ninja and feed them into the baseline models. Due to the limitation of GPU memory, we treated 5,000 instructions as a batch.

Table 7: Efficiency of PalmTree and baselines

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

Table 7 shows the encoding time and throughput of different models when encoding the 107 binaries in `Coreutils-8.30`. From the results, we can make several observations. First, PalmTree is much slower than previous embedding approaches such as word2vec and Asm2Vec. This is expected, since PalmTree has a deep transformer network. However, with the acceleration of the GPU, PalmTree can finish encoding the 107 binaries in about 70 seconds, which is acceptable. Furthermore, as an instruction level embedding approach, PalmTree can have an embedding lookup table as well to store some frequently used embeddings. This lookup table works as fast as word2vec and can further boost the efficiency of PalmTree. Last but not least, from the results we observed that it would be 1.7 to 1.9 times slower when doubling the embedding size.

4.6 Hyperparameter Selection

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

5 RELATED WORK

Representation Learning in NLP. Over the past several years, representation learning techniques have made significant impacts in NLP domain. Neural Network Language Model (NNLM) [4] is the first work that used neural networks to model natural language and learn distributed representations for words. In 2013, Mikolov et al. introduced word2vec and proposed Skip-gram and Continuous Bag-Of-Words (CBOW) models [28]. The limitation of word2vec is that its embedding is frozen once trained, while words might have different meanings in different contexts. To address this issue, Peters et al. introduced ELMo [32], which is a deep bidirectional language model. In this model, word embeddings are generated from the entire input sentence, which means that the embeddings can be dynamically adjusted according to different contextual information.

In 2017, Vaswani et al. introduced transformer [39] to replace the RNN networks (e.g., LSTM). Devlin et al. proposed BERT [9] in 2019, which is a bi-directional transformer encoder. They designed the transformer network using a full connected architecture, so that the model can leverage both forward and backward information. Clark et al. [6] proposed ELECTRA and further improved BERT by using a more sample-efficient pre-training task called *Replaced Token Detection*. This task is an adversarial learning process [13].

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

Instruction representation plays a significant role in binary analysis tasks. Many techniques have been proposed in previous studies.

Instruction2Vec [41] is a manually designed instruction representation approach. InnerEye [43] uses Skip-gram, which is one of the two models of word2vec [28], to encode instructions for code similarity search. Each instruction is treated as a word while a code snippet as a document. Massarelli et al. [26] introduced an approach for function-level representation learning, which also leveraged word2vec to generate instruction embeddings. DeepBindiff [11] also used word2vec to generate representations for instructions with the purpose of matching basic blocks in different binaries. Unlike InnerEye, they used word2vec to learn token embeddings and generate instruction embeddings by concatenating vectors of opcode and operands.

Although word2vec has been widely used in instruction representation learning. It has the following shortcommings: first, using word2vec at the instruction level embedding will lose internal information of instructions; on the other hand, using word2vec at the token level may fail to capture instruction level semantics. Second, the model has to handle the OOV problem. InnerEye [43] and DeepBindiff [11] provided good practices by applying normalization. However, normalization also results in losing some important information. Asm2Vec [10] generates embeddings for instructions and functions simultaneously by using the PV-DM model [20]. Unlike previous word2vec based approaches, Asm2Vec exploits a token level language model for training and did not have the problem of breaking the boundaries of instructions, which is a problem of token level word2vec models. Coda [12] is a neural program decompiler based on a Tree-LSTM autoencoder network. It is an end-to-end deep learning model which was specifically designed for decompilation. It cannot generate generic representations for instructions, thus cannot meet our goals.

Representation Learning for Programming Languages. NLP techniques are also widely used to learn representations for programming languages. Harer et al. [15] used word2vec to generate token embeddings of C/C++ programs for vulnerability prediction. The generated embeddings are fed into a TextCNN network for classification. Li et al. [22] introduced a bug detection technique using word2vec to learn token (node) embedding from Abstract Syntax Tree (AST). Ben-Nun et al. [3] introduced a new representation learning approach for LLVM IR in 2018. They generated conteXtual Flow Graph (XFG) for this IR, which leverages both data dependency and control flow. Karampatsis et al. [17] proposed a new method to reduce vocabulary size of huge source code dataset. They introduced word splitting, subword splitting with Byte Pair Encoding (BPE) [36] cache, and dynamic adaptation to solve the OOV problem in source code embedding.

6 DISCUSSION

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

However, in this paper, we do not intend to learn a language model across multiple CPU architectures. Cross-architecture means that semantically similar instructions from different architectures can be mapped to near regions in the embedded space. Cross-architecture assembly language model can be very useful for cross-architecture vulnerability/bug search. We leave it as a future work.

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

7 CONCLUSION

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

PalmTree can be pre-trained by performing self-supervised training on large-scale unlabeled binary corpora. PalmTree is based on the BERT model but pre-trained with newly designed training tasks exploiting the inherent characteristics of assembly language. More specifically, we have used the following three pre-training tasks to train PalmTree: MLM (Masked Language Model), CWP (Context Window Prediction), and DUP (Def-Use Prediction). We have designed a set of intrinsic and extrinsic evaluations to systematically evaluate PalmTree and other instruction embedding models. Experimental results show that PalmTree has the best performance in intrinsic evaluations compared with the existing models. In extrinsic evaluations that involve several downstream applications, PalmTree outperforms all the baseline models and also significantly improves downstream applications' performance. We conclude that PalmTree can effectively generate high-quality instruction embedding which is helpful for different downstream binary analysis tasks.

8 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by National Science Foundation under grant No. 1719175, and Office of Naval Research under Award No. N00014-17-1-2893. 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] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[2] Amir Bakarov. 2018. A Survey of Word Embeddings Evaluation Methods. *CoRR* abs/1801.09536 (2018). arXiv:1801.09536 http://arxiv.org/abs/1801.09536
[3] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: a learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 3589–3601.
[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

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

[6] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

[7] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 2702–2711.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2978–2988.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

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

[11] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. *NDSS* (2020).

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

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

[14] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. {DEEPVSA}: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1787–1804.

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

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

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

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

[19] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations*.

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

[21] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. 3835–3845.

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

[23] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*.

[24] 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).

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

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

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

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

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

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

[31] 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).

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

[33] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10, 1872–1897. https://doi.org/10.1007/s11431-020-1647-3

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

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

[36] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725.

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

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

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[40] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[41] Lee Young Jun, Choi Sang-Hoon, Kim Chulwoo, Lim Seung-Ho, and Park Ki-Woong. 2017. Learning Binary Code with Deep Learning to Detect Software Weakness. In *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*.

[42] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

[43] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *NDSS*.

A OPCODE AND OPERAND TYPES FOR OUTLIER DETECTION

Table 8 shows how we categorize different opcodes by referring to [29]. Table 9 shows how we categorize different operand types. The first column shows the type of operands combination. "none" means the instruction has no operand, such as **`retn`**. "tri" means the instruction has three operands. The other ones are instructions that have two operands. For instance, "reg-reg" means both operands are registers. The type of each operand has been listed in the second and third columns.

B MORE FIGURES IN EVALUATIONS

Figure 15 and Figure 16 show the results of EKLAVYA in the Function Type Signature Inference task. Figure 15 is the loss value curves

Table 8: Types of Opcodes

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

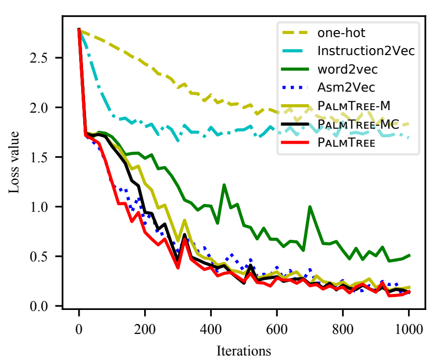

Figure 15: Loss value during training

of EKLAVYA during training. Figure 16 shows the accuracy curves during the training.

Table 9: Types of Operands

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

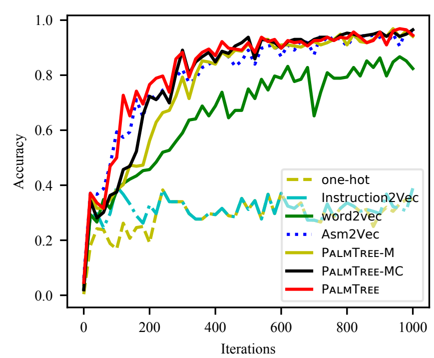

Figure 16: Accuracy during training

C HYPERPARAMETERS

C.1 Embedding sizes

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

Table 10 shows all of the results of intrinsic evaluation when having different embedding sizes. From the results, we can observe that there is a clear trend that the performance becomes better when increasing the embedding size. The largest embedding size has the best performance in all three metrics. However, considering efficiency, we recommend having a suitable embedding size configuration according to the hardware capacities. For example, we only have a single GPU (GTX 2080Ti) in our server, thus we chose 128 as the embedding size.

C.2 Output layer configurations

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

Table 10: Embedding sizes

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

second-last layer, the third-last layer, and the fourth-last layer, on our intrinsic evaluation tasks. The embedding size in this experiment is set as 128.

Table 11: Output layer configurations

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

Table 11 shows all of the results of the intrinsic metrics when having a different layer as the output layer. There is no obvious advantage to choose any layer as the output layer. However, the second-last layer has the best results in opcode outlier detection and basicblock similarity search. Thus we chose the second-last layer as the output layer in this paper.

C.3 Context window for CWP

Table 12: Context Window Sizes

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

In this experiment, we evaluate the performance of PalmTree with different context window sizes in the CWP task. For instance, if the context window size is 2, it means that we consider $n-2$, $n-1$, $n+1$ and $n+2$ as contextual instruction when given instruction n as a sample. We evaluate 1, 2, 3, and 4 as four different context window sizes in this experiment. Table 12 shows all of the results of the intrinsic metrics when training PalmTree with different context window configurations. We can observe that context window size 1 and 2 have similar performance on the three intrinsic evaluation metrics, but context window size 2 has the best performance on the downstream task EKLAVYA. Further increasing the context window size to 3 and 4 will lead to worse results. Based on these results, we choose the context window size to be 2.