Geometric Analysis and Metric Learning of Instruction Embeddings

Sajib Biswas
Department of Computer Science
Florida State University
Tallahassee, Florida
biswas@cs.fsu.edu

Timothy Barao
Department of Computer Science
Florida State University
Tallahassee, Florida
barao@cs.fsu.edu

John Lazzari
Department of Computer Science
Florida State University
Tallahassee, Florida
jcl19h@fsu.edu

Jeret McCoy
Department of Computer Science
Florida State University
Tallahassee, Florida
jdm19a@fsu.edu

Xiuwen Liu
Department of Computer Science
Florida State University
Tallahassee, Florida
liux@cs.fsu.edu

Alexander Kostandarithes
Department of Computer Science
Florida State University
Tallahassee, Florida
ak17c@fsu.edu

2022 International Joint Conference on Neural Networks (IJCNN) | 978-1-7281-8671-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IJCNN55064.2022.9892426

Abstract—**Embeddings for instructions have been shown to be essential for software reverse engineering and automated program analysis. However, due to the complexity of dependencies and inherent variability of instructions, instruction embeddings using models that are successful for natural language processing may not be effective. In this paper, we perform geometric analysis of instruction embeddings at the token level and instruction family level, showing much greater variability and leading to degraded performance on intrinsic analyses. Then we propose to use metric learning to improve the relationships among instructions using triplet loss. Our results on a large dataset of instruction groups shows significant improvements. We also provide a theoretical analysis of the instruction embeddings by looking at the BERT components and characteristics of inner-product matrices for attention in the transformer blocks. The code will be available publicly after the paper is accepted for publication.**

I. Introduction

With recent cyber attacks upon key infrastructures becoming more quotidian, the need for strong cybersecurity to protect these industries has become more important than ever. One major challenge facing security experts is program analysis, the process of understanding a program, given only the binary application. Software reverse engineering can provide invaluable insights about a program's design [1], assisting static analysis of suspicious code and dynamic analysis of such program in a sandbox environment [2]. If done manually, such analysis can consume valuable time and may require significant amount of domain knowledge from the analyst. Recently, deep learning has been proven to be useful in various forms of binary analysis tasks. Research efforts in this direction includes works on function boundary identification [3], malware detection [4], and binary code similarity search [5], [6]. Deep learning based approaches are showing promising results, often better performances than traditional and other machine learning based techniques. Due to the similarities found in the structure of natural language documents and computer programs [7], the similar deep learning techniques that are applicable for natural languages can be used for programs analysis to resolve the existing challenges and achieve more accurate results.

Representation learning [8] has become very popular since it is able to transfer knowledge across multiple language entries, and thus significantly improve the effectiveness and robustness of NLP performance. Automatically learning word and sentence-level representation is desirable because it can capture higher-level semantic features without requiring expert domain knowledge. Recently, pre-trained models for natural language processing [9] are becoming trendy, since training a model from the scratch is becoming expensive as the numbers of model parameters are increasing rapidly. Some state-of-the-art examples of deep pre-trained language models based on representation learning in NLP include BERT [10], GPT [11], and ALBERT [12]. Such a pre-trained model can later be fine-tuned for specific downstream tasks.

Security researchers have been deploying NLP-based techniques for solving the problem of instruction representation learning and capturing the higher-level characteristics of instructions for quite some time now. Existing works based on NLP, such as Asm2Vec [13], and InnerEye [14] use control flow graph (CFG) to capture contextual information between instructions, which can vary based on different compilers and their optimizations. Recently, Li et al. [15] introduced a new pre-trained assembly language model, which is able to generate general-purpose instruction embeddings, based on transformers [16], specifically the BERT model [10], called PalmTree. Unlike the previous models, the goal of PalmTree is to develop a method of representing instructions in a general, compiler-independent fashion. In this paper, we focus on analyzing the PalmTree model. By the geometric analysis of the model's embeddings, it is possible to evaluate the effectiveness of their representations [17]. We also conduct an intrinsic evaluation using the pre-trained model made public by the authors, which is designed to detect an outlier within a group of instructions based on their opcodes. Using metric learning [18], we show that it is possible to improve the performance of the embeddings substantially.

The rest of the paper is structured as follows. First, we discuss some works related to the topic in Section II. In Section III, the subject of geometric analysis of instruction embeddings is discussed, followed by an introduction to Metric Learning and the application of Metric Learning with respect to instruction embedding in Section IV. Finally, the experimental results are shown in Section V. Section VI provides a theoretical analysis of the model to provide explanations of the observed performance and Section VII concludes the paper with a brief summary and future work for improving the PalmTree model and applying NLP-based techniques to software reverse engineering and program analysis.

II. Related Work

In this section, we first discuss several works that focus on implementing representation learning in the field of natural language processing. Then, we review some of the attempts at leveraging NLP-based techniques for representing instructions, aiming for automated program analysis, the rationale behind doing so, and some common challenges that occur due to the inherent characteristics of binary programs.

Bengio et. al. [19] published the first work to use neural networks for natural language modeling by learning a distributed representation for words. Later, word2Vec [20], [21] was introduced in 2013, and proposed Skip-gram and Continuous Bag-Of-Words(CBOW) models. However, due to their limitations, these models fail to capture context-level information.

For a long time, recurrent neural networks were used for processing natural languages due to their sequential nature. In 2017, Transformer [16] was introduced by Vaswani et al. which was proposed as a replacement for RNN models. It implements a self-attention mechanism which relies on capturing the relationships between words in the same sentence. Based on the transformer model, Devlin et al. introduced BERT [10], which stands for Bidirectional Encoder Representations from Transformers. The neural network in BERT leverages a fully connected architecture, which enables the representation to combine the left and the right context together at the same time.

Just like natural languages, programming languages including low-level assembly languages have well-defined syntactic and grammatical rules. So, it is fitting to utilize NLP techniques to facilitate binary program analysis. Instruction2Vec [22] is an approach to produce instruction representation, where the features are manually engineered. There are multiple attempts at instruction representation based on Word2Vec. For example, InnerEye [14], and DeepBindiff [23] leverage word2Vec [20] to generate instruction embeddings to achieve good results in basic block similarity search across different binaries. These models treat each instruction as a single word. Besides these works, Asm2Vec [13] utilizes the PV-DM model [24], which is basically an extension of Word2Vec, to generate instruction embeddings to match semantically similar functions in binary programs. In that work, each instruction is split into one opcode and up to two operands.

Since BERT [10] is one of the state-of-the-art models in the NLP domain, it has been leveraged for program analysis. There have been several works in this regard. For example, Yu et al. [25] takes the control flow graph of a program as input and then uses BERT to pre-train the embeddings of its tokens and basic blocks, which can be leveraged for binary code similarity detection. Trex [26] takes one of BERT's pre-training tasks, namely Masked Language Model (MLM) to learn semantics by leveraging under-constrained dynamic execution to achieve the goal of detecting binary code similarity.

PalmTree [15], which stands for Pre-trained Assembly Language Model for InsTRuction EmbEdding, is one of the most recent attempts at leveraging BERT for learning instruction embeddings with the goal of facilitating binary analysis. PalmTree provides a pre-trained assembly language model and tries to solve the unique challenges that come with adopting learning-based encoding approaches to model instructions. To tackle the complexity and heterogeneity in instruction formats, PalmTree focuses on getting the internal details about each instruction. Also, getting contextual information from control flow alone can get noisy due to compiler optimizations. So, PalmTree throws data flow dependency into the mix. The authors of PalmTree made a pre-trained model available to the public. We use this pre-trained model to design and conduct all our experiments.

III. Geometric Analysis of Instruction Embeddings

A. Instruction Embeddings

In BERT, to generate a representation of a sentence, word embeddings are fused with their positional information, and then fed as input to the model for training. BERT employs two types of pre-training tasks, namely Masked Language Model (MLM), and Next Sentence Prediction (NSP). The MLM objective enables the representations to learn the words' context bidirectionally to contextualize the relationships between them. NSP is used to pre-train sentence-pair representations to capture the semantic relation between sentences in a document.

For instruction embeddings, the basic idea is very similar. Unlike previous works [13], [14], [23], PalmTree deconstructs each instruction into a variable number of tokens. Here, each instruction is considered to be a sentence and the tokens are comparable to words. For example, given an instruction "**mov rbx, dword [rsp+0xb8]**", PalmTree decomposes it into the following tokens: "**mov**", "**rbx**", "**dword**", "**[**", "**rsp**", "**+**", "**0xb8**", and "**]**".

Like BERT, PalmTree [15] combines each token's embedding with their positional information. For pre-training, it utilizes MLM the same way. To capture control flow information, PalmTree trains instructions in pairs. However, it deploys context window prediction (CWP) in place of NSP, since the same strictness may not be suitable for capturing contextual information in the control flow of assembly languages. The authors themselves introduce a third pre-training task called

Def-Use Prediction (DUP), which predicts if two instructions have a def-use relation, and thus includes data dependency information in their assembly language model.

B. Geometric Properties of Token-level Embeddings

BERT has set new standards for state-of-the-art in various NLP benchmarks, but it is still not completely understood why BERT works so well. There have been several attempts by researchers to understand BERT by analyzing the geometry of its word embeddings [17], [27], [28]. The focus of these works is on the characteristics of the learned representations rather than the patterns learned by the attention mechanism. Results show that, while BERT is better than most at capturing the notion of similarity in terms of analogy query, the embeddings do not show very well clustering in the vector space based on semantics.

Since each token is considered a word after decomposition, their embeddings are expected to show behaviors demonstrated by word embeddings from state-of-the-art language models such as BERT. For example, we can expect them to display fairly strong pairwise correlations as Podkorytov et al. [17] demonstrated.

To this end, we start our analysis by first looking at the geometric properties of token-level embeddings produced by the pre-trained model of PalmTree. We computed the pairwise correlations of weights of each of the tokens this pre-trained model generates. Here, the size of the vocabulary is 6631. The pre-trained PalmTree model generates 128-dimensional vectors for each of the tokens in it's vocabulary. Hence, computing the correlations produces 6631×6631 values in the $[-1, 1]$ range. The histogram showing this correlations is shown in Figure 1. Since the plot is centered around 0, it can be said to be similar to a normal distribution, which implies the token-level embeddings do not show strong pairwise correlations between them.

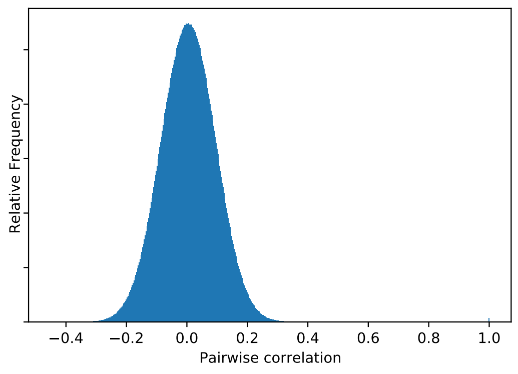


Fig. 1: Pairwise correlations between the weights of the token embeddings in PalmTree.

C. Intra- and Inter-group Analysis of Instruction Embeddings

To analyze the geometric properties of the instruction embeddings, we utilize the dataset used by PalmTree for its intrinsic evaluation. The dataset consists of 7703 instructions where all the instructions are categorized into 10 groups based on their opcode. The different categories along with the total number of instructions in each category and some examples are shown in Table I. Each instruction from every group was encoded by PalmTree to get it's embedding representation. We then calculated the intra-group cosine distances, as well as the inter-group cosine distances for all of the instructions' embeddings.

TABLE I: Brief description of the ten different categories of instructions in the PalmTree model, based on their opcodes.

Op-code	Num. of instructions	Example
MOV	4206	mov,r11,rdx
BINOP	1743	sub,r8,rbp
CALL	25	call,eax
CMP	1272	cmp,r14,address
JMP	40	jmp,ecx
SHIFT	153	shl,eax,0x10
CSET	44	setne,al
CMOV	192	cmovae,r10,rdx
UNARY	25	inc,eax
FP	3	faddp,st1
Total	7703	

We present the results of our analysis in Figure 2. Overall, it appears that there is no significant difference between the groups, as the curves for the aggregate values are similar, almost overlapping with each other. Even in the plots that show the distributions of the intra-family and inter-family cosine distances for a particular family of instructions, the distances are still large, and few instructions seem to be close/similar.

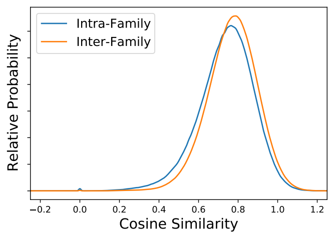
These observations led us to conclude that the embeddings of the instructions do not display similarity that is representative of their semantic relations, such as whether they belong to the same group or come from different groups of instructions.

IV. Metric Learning for Instruction Embeddings

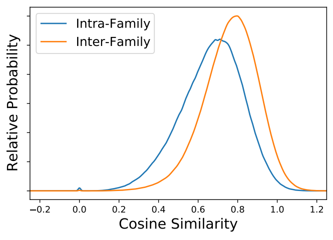
The goal of metric learning is to train a neural network $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ to map inputs $x \in \mathbb{R}^n$ to a m dimensional metric space. The distances between these embeddings should then be small when the inputs share the same label, and larger when they do not. The distance function defined on the metric space, such as Euclidean distance, is then used to evaluate the loss and train the parameters for this purpose. In this case, instruction embeddings are passed through the neural network and further separated using triplet loss, where outliers correspond to negative samples.

Let X be a dataset consisting of groups of instruction embeddings g. For each $g \in X$, we define the triplets as $\mathcal{T}_g = \{(s_a, s_p, s_n)_1, ..., (s_a, s_p, s_n)_N\}$, where N is the possible number of triplets formed by the elements of the group and $s_a, s_p, s_n \in g$. In this case, s_a and s_p are correspond to similar instructions, and s_n is the outlier in the group. The goal is to minimize triplet loss, defined as

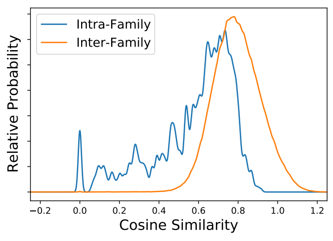
$$L_{triplet} = \frac{1}{N} \sum_{\mathcal{T}_g} [\|f(s_a) - f(s_p)\|_2 - \|f(s_a) - f(s_n)\|_2 + \alpha]_+,$$



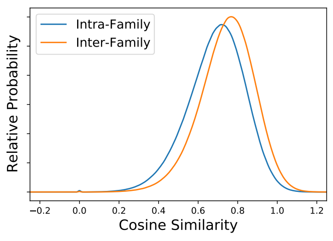
(a) BINOP Family



(b) CMP Family



(c) JMP Family



(d) Aggregated over all family of instructions

Fig. 2: Cosine distances of the instructions embeddings within each of the instruction families (intra-family) and also between the ones in the family and all the ones in the other families (inter-family). While (a), (b) and (c) show the probability distributions for the three representative families, (d) is the average of all the probability distributions in the entire dataset.

which minimizes the distances between the positive samples and anchors and maximizes the distances between the anchors and outliers using the mean reducer [29]. The criterion is satisfied once $\|f(s_a) - f(s_p)\|_2 + \alpha \leq \|f(s_a) - f(s_n)\|_2$, meaning the Euclidean distances are fully separated by the margin parameter α. We create and train a deep metric learning model to improve the instruction embeddings of the PalmTree model and test it by conducting an intrinsic evaluation based on the concept of outlier detection. The details of the experiments are described in Section V.

V. Experimental Results

A. The Dataset

For fairness, we use the same dataset used by the PalmTree authors for their intrinsic evaluation. The dataset we use for the experiments comes in two parts. The first part consists of all the tokens' embeddings in the vocabulary and another one encompasses all of the instructions, grouped into several categories based on their opcodes.

The size of the vocabulary for which we conduct experiments on token-level embeddings is 6631. Each instruction, symbol, register, even some constants are assigned specific tokens. Some special tokens are used as placeholders to alleviate the out-of-vocabulary problem.

For conducting the evaluation on the instructions' embeddings, we use the dataset summarized in Table I.

B. Analysis

As discussed earlier, the embeddings for the instructions are essential in order for the models built on them to generalize well. Here we first show the results of our analysis on the embeddings of the tokens and then on that of the instructions. Then we show the performance on an intrinsic evaluation with and without metric learning.

1) Token-level Embeddings: It is already shown in Figure 1 that the distribution of pairwise correlations between the tokens' embeddings resembles that of a normal distribution. In this section, we take a particular example token, compute the cosine similarity of all the tokens in our vocabulary and then sort the tokens in the descending order of their similarity to our target token. This way, we can take a look at the tokens which are the closest to our target in terms of their similarity measure.

For example, in Figure 3, we plot the pairwise cosine similarities of the top 20 tokens' embeddings in relation to the token 'add'. As expected, the token 'add' has the highest similarity with itself. But, if we look at the other tokens, we do not see any tokens that are semantically related to our target token 'add'.

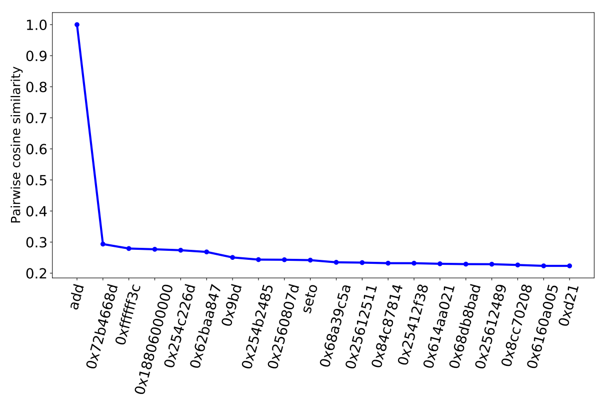


Fig. 3: Pairwise cosine similarities of tokens' embeddings that are most similar to the token 'add'.

We pick another token '0x1', which is an operand, do the same analysis, and plot the result in Figure 4. Unsurprisingly, the results are very similar to that of Figure 3. The token's embedding only seems to be related to itself and most of the tokens in the top 20 are not relevant. While tokens such as '0xa3' and '0x1a' are semantically related to our target token, we also see 'cmovl', which is actually an opcode and is not closely related to '0x1'

We also evaluate the lengths of embeddings for the tokens and plot their distribution in the form of a histogram in Figure 5. The distribution centers around the mean, which is 10.29. Compared to the embeddings of the pretrained BERT models,[1] the embeddings for PalmTree are much longer. In addition, these embeddings are much longer than the inner

[1]See https://huggingface.co/docs/transformers/model_doc/bert for pretrained BERT models.

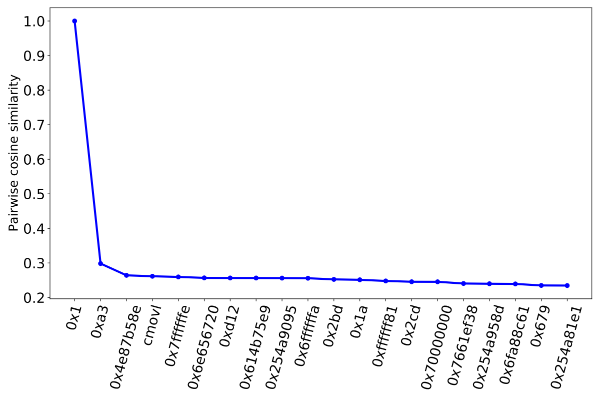


Fig. 4: Pairwise cosine similarities of tokens' embeddings that are most similar to the token '0x1'.

products between two different token embeddings, which have a negative effect of the instruction embeddings due to limited mixing of tokens; see Section VI for in-depth analysis.

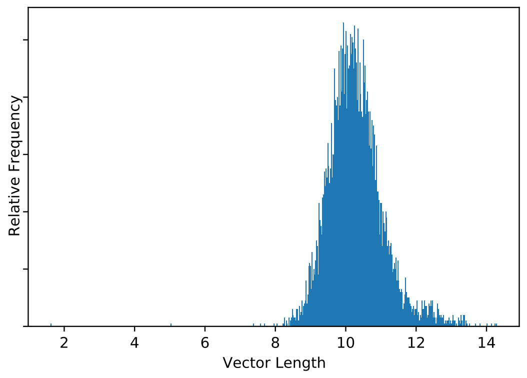


Fig. 5: Distribution of the length of token-level embedding vectors for all of PalmTree's vocabulary.

2) Instruction Embeddings: We also conduct similar experiments for the instruction embeddings generated by the pre-trained PalmTree model. We pick a particular instruction, then compute its cosine similarity with all the instructions' embeddings in our dataset and sort the instructions in decreasing order of their pairwise cosine similarities with our target instruction. After that, we plot the topmost 20 cosine similarity values along with the instructions they represent. As an example, we pick the instruction 'or, ecx, r10d', which belongs to the BINOP group according to Table I, then we plot the pairwise cosine similarities of the top 20 instructions' embeddings in relation to this target instruction. The result is shown in Figure 6. Here, unsurprisingly the instruction shows the highest similarity with itself. If we look at the other instructions along the X-axis, we can see that although they mostly include a binary operator in them (e.g., xor, sub, add etc.), there is a 'mov' instruction within the group. This suggests that while instruction embeddings generated by PalmTree actually display better semantic relations between them than token-level embeddings, a high level of inaccuracy exists there.

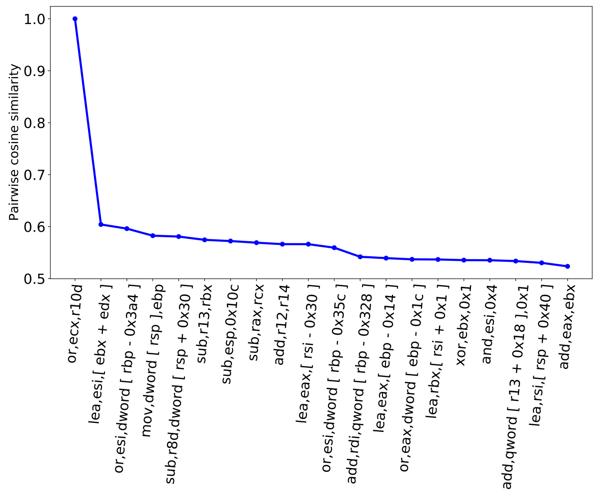


Fig. 6: Pairwise cosine similarities of instructions' embeddings that are most similar to the instruction 'or, ecx, r10d'.

C. Intrinsic Evaluation and Metric Learning

In the field of NLP, intrinsic evaluation refers to determining the quality of the generated embeddings by examining how they perform on designed datasets. We conduct the *instruction outlier detection* originally designed by the authors of the PalmTree model. For this, we use the dataset already summarized in Table I. First, we create a set of randomly chosen instructions, one of which is an outlier. The basis on which an instruction is considered an outlier is its opcode. To find out which one is the outlier, we calculate the pairwise cosine distances among all the instructions' embeddings within the set of instructions, and chose the one which is the furthest from the rest. A total of $50,000$ instruction sets are created for the evaluation. Each set consists of 4 instructions from the same opcode category and 1 from a different category.

First, we conduct the intrinsic evaluations using the pre-trained PalmTree model to see how the model actually performs. For efficiency, we compute the embeddings of every instruction in the dataset beforehand, and store them to use later for computing cosine distances. Then, we conduct the intrinsic evaluations on 10 different instances of $50,000$ set of instructions and compute the accuracy for each instance. Here, the term *accuracy* is defined as the percentage of correct detection of the outlier in a set of 5 instructions, based on the pairwise cosine distances of their embeddings. For the PalmTree pre-trained model, the mean accuracy for all 10 instances are nearly 68%. This value is much lower than the accuracy originally reported in the PalmTree paper [15]. A main reason is how the instructions were chosen in their evaluation. While they require the opcode and the number of operands to match for the instructions, we choose instructions in a set solely based on their opcode, without considering the number of operands they consists of. Therefore our sets have higher variability and are more difficult to detect outliers reliably.

We construct a deep metric learning model based on the concepts we discussed in section IV to conduct the same experiment and show off the improvements in the instructions' representations (i.e., embeddings) made by the application of metric learning. Our embedding network is comprised of two linear layers containing 256 neurons with ReLU activation. The network embeds the instructions to a 128-dimensional vector where triplet loss pushes the outliers further from the similar instructions. We train our network using the SGD optimizer with a learning rate of 0.001 and momentum of 0.9 for 10 epochs. Usually, a mining technique such as hard online negative mining or batch hard mining [30] would be needed to sample relevant triplets so the model can converge faster. In our case, our model quickly learns the important features needed to distinguish between similar instruction embeddings and outliers, thus mining tactics are not needed.

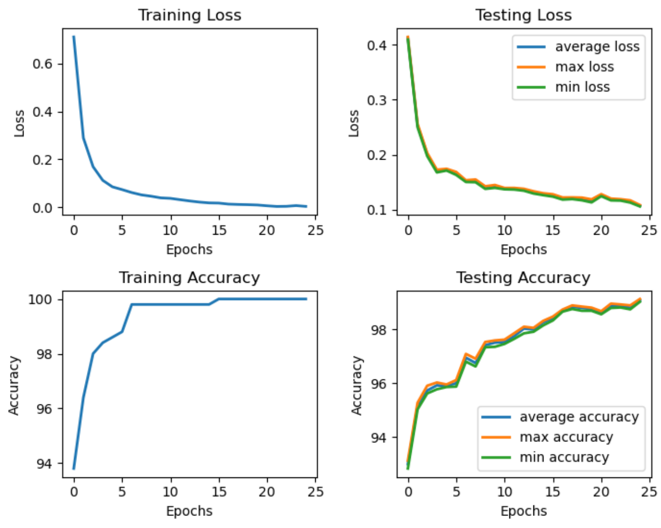


Fig. 7: Loss and accuracy curves during training. The accuracy was taken as the average of correct outliers detected using 10 instances of $50,000$ randomly generated groups. We also display the maximum and minimum accuracy and loss over the 10 instances, and find they do not vary much.

When training on only 500 groups for 25 epochs, we were able to achieve an accuracy of 99% on $49,500$ testing groups. When evaluated using the same $49,500$ testing samples, the original embeddings (generated by PalmTree) only achieve an accuracy of 68%. We conduct various experiments testing the speed of convergence of the model as well as its accuracy when trained using more than 500 groups, and find that the model is able to achieve 100% accuracy as the size of the training set grows. In general, we see that using metric learning drastically improves the relationship between instruction embeddings from the same group, while further separating the outliers. The loss and accuracy for both training and testing are shown in Figure 7.

D. Comparisons and Discussions

To visualize the effects of applying metric learning, we plot the t-SNE [31] of the datset before and after the application of metric learning, which is shown in Figure 8. One thing is instantly evident that, although we have 10 categories based on opcode, the distribution is not very well-balanced. For example, while the group for opcode 'MOV' consists of $4,206$ instructions, another opcode 'CALL' contributes to a small group of only 25 instructions. Now, before applying metric learning, the clusters of instruction embeddings are not distributed properly, and we cannot picture a proper boundary between any two groups. From the second part of Figure 8, we can see that the instructions show well-formed clusters according to their category. We can easily separate one group from another after applying metric learning on the instructions' embeddings.

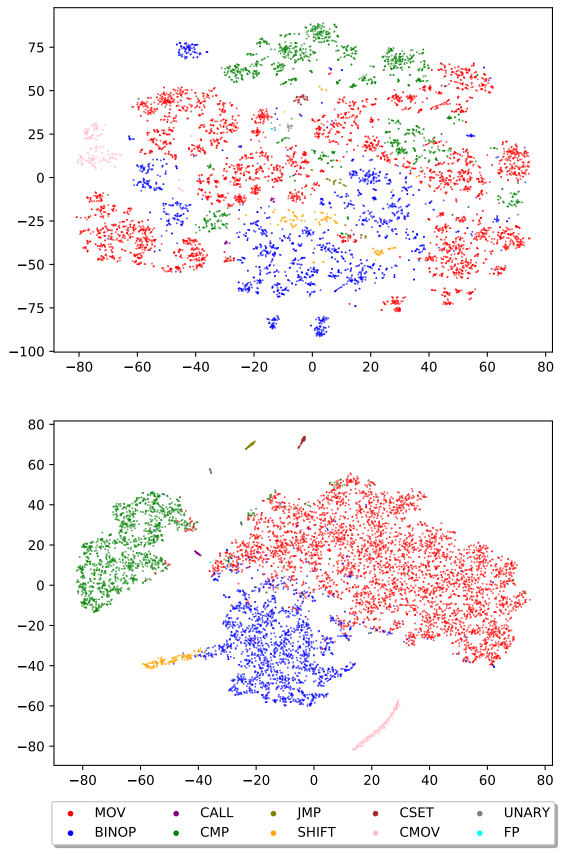


Fig. 8: The t-SNE representation of data before (top) and after (bottom) metric learning. Instruction pairs that are in the same class are closer to each other. Triplet loss has been used to determine similarity between instructions throughout the clusters.

VI. Theoretical Explanations

In this section, we aim to explain the empirical results shown in this paper that have been reported by analyzing the BERT architecture. The BERT used in the PalmTree model is a standard stack of transformer blocks, which can be described compactly by a set of equations.

A transformer block is a parametrized function class $f_\theta : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$. If $\mathbf{x} \in \mathbb{R}^{n\times d}$ then $f_\theta(\mathbf{x}) = \mathbf{z}$; letting $Q^{(h)}(\mathbf{x_i}) = W_{h,q}^T\mathbf{x}_i$, $K^{(h)}(\mathbf{x_i}) = W_{h,k}^T\mathbf{x}_i$, $V^{(h)}(\mathbf{x_i}) = W_{h,v}^T\mathbf{x}_i$, where $W_{h,q}$, $W_{h,k}$, $W_{h,v}$ are all in $\mathbb{R}^{d\times k}$, the function is implemented using a sequence of transformations. For the BERT in the PalmTree model, $d = 128$ and $H = 8$. Equation 1 computes the weights used for self attention,

$$\alpha_{i,j}^{(h)} = \texttt{softmax}_j\left(\frac{\left\langle Q^{(h)}(\mathbf{x}_i), K^{(h)}(\mathbf{x}_j)\right\rangle}{\sqrt{k}}\right). \quad (1)$$

Note that the softmax is applied row wise; since softmax is scale sensitive, when the difference between the largest and second largest dot products in each row is large, the output probability distribution will become very close to a delta function.

$$\mathbf{u}_i' = \sum_{h=1}^{H} W_{c,h}^T \sum_{j=1}^{n} \alpha_{i,j} V^{(h)}(\mathbf{x}_j), \quad W_{c,h} \in \mathbb{R}^{k\times d}, \quad (2)$$

Equation (2) implements the self attention via weighted averages, where the ones with larger weights affect the output more and therefore the model "pays" more attention. The output is generated by applying token-wise layer normalization, forward transformation, and another layer normalization, given by equations (3), (4), and (6),

$$\mathbf{u}_i = \texttt{LayerNorm}(\mathbf{x}_i + \mathbf{u}_i'; \gamma_1, \beta_1), \quad \gamma_1, \beta_1 \in \mathbb{R} \quad (3)$$

$$\mathbf{z}_i' = W_2^T \texttt{ReLU}\left(W_1^T \mathbf{u}_i\right), \quad W_1 \in \mathbb{R}^{d\times m}, W_2 \in \mathbb{R}^{m\times d}, \quad (4)$$

$$\mathbf{z}_i = \texttt{LayerNorm}(\mathbf{u}_i + \mathbf{z}_i'; \gamma_2, \beta_2), \quad \gamma_2, \beta_2 \in \mathbb{R}, \quad (5)$$

$$\texttt{LayerNorm}(\mathbf{z}; \gamma, \beta) = \gamma \otimes \frac{\mathbf{z} - \mu_{\mathbf{z}}}{\sigma_{\mathbf{z}}} + \beta, \quad (6)$$

where $\mathbf{z}, \gamma, \beta \in \mathbb{R}^d$, $\mu_{\mathbf{z}}$ is the mean of the mini-batch and $\sigma_{\mathbf{z}}$ is the estimated variance during the training time.

Here equations (1) and (2) implement the self-attention, where the embeddings of the entire sequences are mixed, the importance of different tokens is based on the inner products of the query embedding and that of the current token. Note the attention mechanism is scale-sensitive. In other words, if we scale all the token embeddings by a constant larger than one, the mixing effect will be reduced as the probability distributions given by the softmax will be more peaked. For the PalmTree model [15], we have checked the diagonal elements and off-diagonal elements before softmax is applied in equation (1). We have realized that the diagonal elements are much larger; for the average diagonal and off-diagonal elements in the input layer are 98.53 and 1.45 respectively and they are 3733.10 and −112.15 respectively for the output layer. The inner products for the input layer are also consistent with the token lengths shown in Fig. 5.[2] In all the instructions we have examined, the diagonal elements are much larger than the rest in each row. This shows that the BERT model used does not provide a strong context-sensitive representation for an instruction. In addition, the embedding chosen for an instruction is the average from the second last layer.

While the two layer-norm layers and the feed forward layer are important, they are uniformly applied to all the embedding vectors. While the directions could change, embeddings do not affect each other in these layers.

In addition, the additional loss terms (CWP and DUP loss terms) use the output vector associated with the special [CLS] token, but the instruction embedding is the average for the second last layer. The gradients from those loss terms may not affect the instruction embeddings as expected. Furthermore, the interactions among the three loss terms need to be examined more carefully.

We believe the limited mixing of the token representations along with the relative low correlations between the embeddings any pair of input tokens affects the effectiveness of the resulting instruction embeddings. As shown in Fig. 2, cosine distances between instruction embeddings within a family overlap heavily with instruction embeddings from two families; in some cases they are almost identical. The overlaps limit the inherent ability of the instruction embeddings to detect outliers reliably. While metric learning improves the performance significantly, more effective instruction embeddings should further improve the robustness and generalization performance. However, how to further improve the robustness and effectiveness of the token and instruction embeddings is being investigated further.

VII. Conclusion and Future Work

In this paper, we have systematically analyzed the embeddings for instructions by looking at the geometric relationships among the vectors in the embedding space. Our results show that the embeddings of the tokens and instructions do not reflect the similarities of tokens and instructions robustly, resulting in low performance on an outlier detection problem. Metric learning using triplet loss is effective in mitigating the issues, resulting in better clusters and significant improvements for the outlier detection problem. As the embeddings are essential for downstream applications, how to improve them remains to be an important problem.

More broadly, due to the many parallels between computer programs and natural languages, and the importance of linking high level concepts and explanations in natural languages to low level constructs (basic blocks, loops, functions, and so on), how to take advantage of the recent development in natural language processing to improve efficiency and accuracy of software reverse engineering, malware analysis, and program analysis should be explored systematically. In particular, by utilizing the data for programs, transfer learning techniques can be an effective direction, which is being investigated.

[2]Note that the length of an embedding vector is the square root of the inner products of the vector and itself.

Acknowledgement

We like to thank the authors of the PalmTree paper [15] for sharing their code, which helps us to conduct the intrinsic evaluation. This research was partially supported by NSF grant DGE 1565215.

References

[1] A. K. Gahalaut and P. Khandnor, "Reverse engineering: an essence for software re-engineering and program analysis," *International Journal of Engineering Science and Technology*, vol. 2, no. 06, pp. 2296–2303, 2010.

[2] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 94–109.

[3] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 611–626.

[4] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

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

[7] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[8] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[9] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[11] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[12] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *arXiv:1909.11942*, 2019.

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

[14] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

[15] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3236–3251.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

[17] M. Podkorytov, D. Bis, J. Cai, K. Amirizirtol, and X. Liu, "Effects of architecture and training on embedding geometry and feature discriminability in bert," *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2020.

[18] D. Yi, Z. Lei, S. Liao, and S. Z. Li, "Deep metric learning for person re-identification," in *2014 22nd international conference on pattern recognition*. IEEE, 2014, pp. 34–39.

[19] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," in *J. Mach. Learn. Res.*, 2003.

[20] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR*, 2013.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013.

[22] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *KSII the 9th international conference on internet (ICONI) 2017 symposium*, 2017.

[23] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Security Symposium*, 2020.

[24] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[25] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[26] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[27] E. Voita, R. Sennrich, and I. Titov, "The bottom-up evolution of representations in the transformer: A study with machine translation and language modeling objectives," in *arXiv:1909.01380*, 2019.

[28] K. Ethayarajh, "How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings," in *EMNLP/IJCNLP*, 2019.

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

[30] A. Hermans, L. Beyer, and B. Leibe, "In defense of the triplet loss for person re-identification," *arXiv preprint arXiv:1703.07737*, 2017.

[31] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.