GENTAL: GENERATIVE DENOISING SKIP-GRAM TRANSFORMER FOR UNSUPERVISED BINARY CODE SIMILARITY DETECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Binary code similarity detection serves a critical role in cybersecurity. It alleviates the huge manual effort required in the reverse engineering process for malware analysis and vulnerability detection, where often the original source code is not available for analysis. Most of the existing solutions focus on a manual feature engineering process and customized code matching algorithms that are inefficient and inaccurate. Recent deep learning based solutions embed the semantics of binary code into a latent space through supervised contrastive learning. However, one cannot cover all the possible forms in the training set to learn the variance of the same semantics. In this paper, we propose an unsupervised model aiming to learn the intrinsic representation of assembly code semantics. Specifically, we propose a Transformer-based auto-encoder like language model for the low-level assembly code grammar to capture the abstract semantic representation. By coupling a Transformer encoder and a skip-gram style loss design, it can learn a compact representation that is robust against different compilation options. We conduct experiments on four different block-level code similarity tasks. It shows that our method is more robust compared to the state-of-the-art.

1 INTRODUCTION

Reverse engineering is the process of analyzing a given binary program without its source code. It routinely requires experienced analysts and demands a huge amount of manual effort. This process is essential in many critical security problems, such as malware analysis, vulnerability discovery, and Advanced Persistent Threat (APT) tracking. Binary similarity detection is an important solution to reduce the amount of manual effort, by detecting known parts and pieces in the target under investigation. Binary-level similarity detection is more difficult than source-level similarity detection, because most of the semantic-rich literals and structures, such as constants, function names, and variable names, are altered or no longer available in the form of assembly language. The data structures are also lost, due to the compilation process, since the debugging information is typically stripped in commercial-off-the-shelf programs.

Many existing approaches rely on manual feature engineering to model the semantics of assembly code. For example, a fragment of assembly code can be modeled into a numeric vector based on (1) the ratio of algorithmic operations, (2) the ratio of transferal operations, and (3) the ratio of function calls [1, 2, 3, 4]. Alternatively, assembly code can be modeled as word-based n-grams [5]. These approaches cannot capture the rich semantics carried in the assembly code. Some other approaches rely on symbolic constraint solving to measure the logical relationship between each pair of code fragments [6, 7]. However, these methods are computationally expensive and do not scale well.

In comparison, recent deep learning approaches have shown to be more effective and robust to detect similar binary code. Typically, a neural network model is proposed and coupled with a contrastive loss function [8, 9, 10, 11]. The network is trained with limited pairs of assembly code with 0-1 labels. Asm2Vec [12] extends the idea of Word2Vec [13] and PVDM [14], and follows an unsupervised paradigm to learn the function representation.

Although deep learning has been proven to be effective, there are still many practical barriers that prevent the aforementioned approaches from a wide adoptation. First of all, most approaches

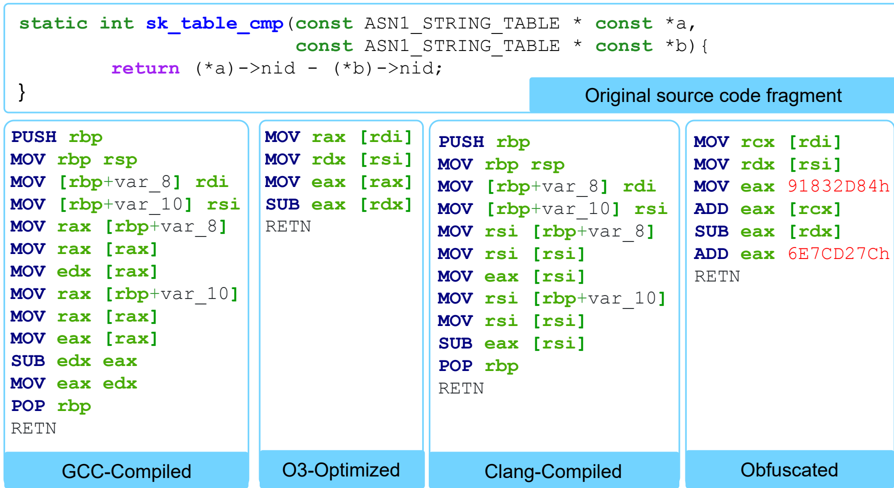

Figure 1: Assembly code blocks compiled from the same source code *sk_table_cmp* from *OpenSSL* with different options. The box on the top shows the source code. From left to right: GCC with O0 optimization level; GCC with O3 optimization level; Clang with O0 optimization; and Clang with O0 optimization and obfuscation.

except Asm2Vec uses a supervised paradigm and learns the model parameters by directly decreasing the distance between similar assembly code pairs in the training set. While supervised models potentially provide better performance in their trained tasks, they usually suffer when a new task is introduced. For example, [10] learns the tasks for cross-optimization level detection and cross-architecture detection on ARM and X86-64. However, it does not guarantee cross-compiler and cross-OS binaries performance, thus it lacks robustness. On the other hand, Asm2Vec, which uses an unsupervised approach, is mainly based on PVDM. This limits its stability during the inference, since it needs multiple rounds to accumulate the gradient. Moreover, Word2Vec based models are less context-dependent than the state-of-the-art models such as BERT [15]. They carry less semantics and discriminative power for the downstream tasks. To illustrate this, Fig. 1 shows 4 pieces of compiled binary code in assembly code, all generated from the same source code. The choice of compilers and optimization levels mainly results in performance differences, while the obfuscated code can be generated for various reasons, both benign and malicious. The semantics of the programs are similar or the same, but the code itself can look drastically different from one another. A supervised model may be able to learn to detect similar binaries cross-compiled, but changing the optimization level or adding obfuscation can create unseen patterns, leading to detection failures.

To learn a compact representation of assembly code through an unsupervised language model, one can use BERT and other Transformer-based models, since they have shown their effectiveness in modeling natural language semantics. However, assembly code, despite the fact that it is missing many semantic-rich literals present in the original source code, complies with a more static syntax. The original Transformer architecture, with the flattened sequence of position-embedded subwords, cannot address the position invariance of the assembly code instructions (see Figure 1). The same instructions may be placed in a different place, but contribute to similar program semantics. Additionally, the original masked language model in BERT distribute the memory of reconstructing the masked tokens into hidden layers of different timestamps. This is against our goal, which is to have a single compact vector representation for all code.

With the above observations, we propose GenTAL, a generative denoising skip-gram Transformer for assembly language representation learning. It follows an unsupervised learning paradigm for binary code similarity detection. We model assembly code instructions through the Transformer encoder in a way that their syntax is preserved. Inspired by the denoising autoencoders, we propose to combine the originally time-distributed memory of the masked language model into a single dense vector, and leverage a skip-gram like structure for masked instructions recovery. This allows the model to embed the semantics of the assembly code into a very compact representation for a more effective similarity detection. Our contributions are as follows:

- We propose a new Transformer-based unsupervised language model for assembly code instructions, the form of human-understandable binary code. The model follows the syntax of assembly code and is able to address the instruction position invariance issue.
- We proposed to combine skip-gram style reconstruction loss with the masked language model to condense the originally time-distributed memory into a single compact embedding vector. This design simulates a denoising autoencoder and provides a unified representation of semantics.
- We conduct experiments on five different scenarios for code similarity detection and compare our methods against traditional TFIDF based and state-of-the-art machine learning-based methods. We show that GenTAL is more robust and able to outperform the baselines in all applications.

2 Related Work

Binary Code Similarity. Assembly code can be regarded as a natural language in certain aspects. For this reason, NLP techniques are often used as encoders. Other ML-based binary code similarity detection works use different approaches for representation learning. Supervised learning approaches, such as Gemini [8], Diff [9], [10], and BinDeep [16] all use siamese networks to reduce loss and use cosine distance to compute similarity. Gemini manually extracts block features and feeds them into a graph neural network. Diff feeds raw byte input into CNN for learning function embedding, which lacks the modeling of block semantics. Yu, et al. [10] extend the BERT model for code semantics learning by introducing the same graph prediction task (SGP) and graph classification task (GC). They also train a graph neural network for assembly code representation learning and a CNN on the adjacency matrix for additional order-aware embedding. BinDeep uses Instruction2Vec [17] with LSTM for instruction and blocks embedding to enable sequence-aware modeling. Asm2Vec [12] follows an unsupervised paradigm and uses PVDM for block embedding. Other traditional graph matching methods include BinDiff [18] and Binslayer [19]. These use static graph patterns. As a result, the performance can be severely hindered with any changes in graphs, which often happen with different compiler settings and obfuscation.

Unsupervised Language Models. BERT [15] is the state-of-the-art language model for NLP pre-training based on Transformer [20] architecture. Transformer can learn the contextual and sequential information within a sentence, while also maintaining multiple levels of semantics. It trains in parallel and is thus faster compared to RNN-based models. There are several variations of the original BERT model. ELECTRA [21] builds on top of BERT and adds a discriminator that predicts which tokens are originally masked. Albert [22] achieves similar performance to BERT, while using fewer parameters through factorized embeddings parameterization and cross-layer weight sharing. RoBERTa [23] further up-trains BERT with heavier parameters and discards the next sentence prediction (NSP) task from BERT.

Word2Vec is also an unsupervised learning technique for language models, which uses Skip-gram or Continuous-Bag-Of-Words (CBOW) to learn word embedding based on a fixed length of windowed context. Doc2Vec extends Word2Vec and adds another ID token for document/sentence representation with the Distributed Memory (PVDM) and Distributed Bag-Of-Words (DBOW) variance. Tokenization is also an important task in language models, since the performance can vary significantly depending on the quality of the tokenizers. There are different levels of tokenization, such as character level and subword level. Byte Pair Encoding (BPE) [24, 25] generates subword vocabulary by learning the frequency of characters in a large corpus. Unigram [26] learns the language model by optimizing the word occurrence given a sequence and then builds the vocabulary by sorting the loss of subwords. These methods are used to combat the out-of-vocabulary (OOV) issues, which many large-scale language models can struggle with.

3 Generative Denoising Skip-gram Transformer

In this section, we describe the details of GenTAL with respect to three major components: preprocessing, including instruction masking and encoding; the code fragment encoding using Transformer; and the reconstruction loss through a skip-gram style approach. The overall framework of GenTAL is shown in Fig. 2. Given a sequence of assembly instructions, which is generated from disassembling the binary data into assembly code, we preprocess the assembly code first to extract the instructions as subword sequences and apply masking. Then each subword is mapped into subword embedding and coupled with an instruction-level positional embedding. After being merged as the instruction

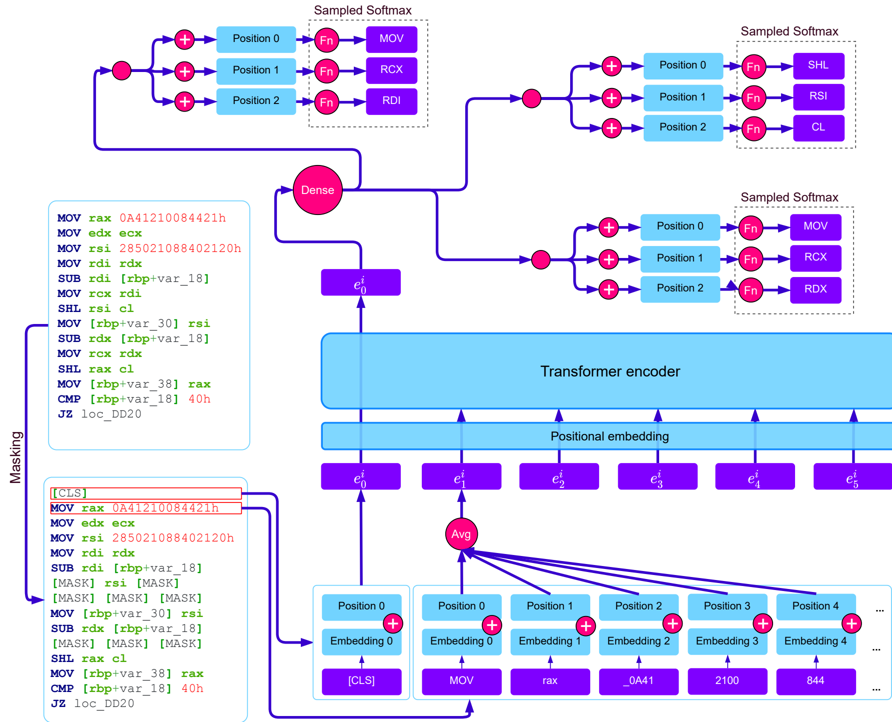

Figure 2: GenTAL's model architecture for training. Given the assembly instructions and their subwords, it first masks certain assembly instructions with a probability. The masked sequences are fed into token embedding and position embedding, in which the results are aggregated to form instruction embeddings. The Transformer then encodes the instructions, where we extract the first `cls` hidden layer output and use it to reconstruct the mask instructions, following a skip-gram style prediction design.

level embeddings, we feed them into a Transformer encoder and obtain the `CLS` step hidden layer output, which is finally position-encoded again to recover the masked instructions. We formally define GenTAL's goal as to learn an encoding function G that maps a sequence of instructions into a b-dimensional space $C \to \mathbb{R}^b$ where its semantics will be preserved by optimizing:

$$\sum_j \sum_i P(t_i | p_j, G(C)) \quad (1)$$

Here C is the sequence of assembly code, p_j is the position of the j-th masked instruction, and t_i is the i-th subword of that j-th instruction. Simply put, we reconstruct the full instruction rather than individual disconnected subwords or the original tokens.

3.1 Instruction Preprocessing, Masking, and Ragged Tensor

An assembly instruction contains an operation and operands. We first treat each instruction as a plain text sequence and clean up the assembly code by removing address-specific literals such as `loc_DD20`. These literals depend on the base address of the binary and can be changed when generated in a slightly different environment. Next, we pre-tokenize the assembly code following the syntax of the assembly language as this can help us alleviate the out-of-vocabulary (OOV) issues. For example, instead of treating `[rbg+var_19]` as a single token, we break it into `[, rbg, +, var, 19]`. After, we train an unigram [26, 27] model on the training corpus for subword tokenization. This can further mitigate the OOV issue, especially on long constants. For example, `0x00fffff1` will be broken down as `0x00`, `ffff`, and `f1`. If any compiler or obfuscator manipulates this constant by shifting or padding, the subwords can still be matched.

Given a sequence of assembly instructions in a linear order, we follow the Masked Language Model used in BERT. However, instead of masking individual subwords or subwords that constitute an original token, we mask a full instruction, since there exists a strong correlation among subwords presented in a single instruction. For example, operand `PUSH` will very likely be followed by a stack register. `XOR` is also used quite often with the same register as its two operands. This issue will make the reconstruction task too easy. Instead, we mask the complete instruction.

Unlike many approaches that treat assembly instructions as a flat sequence of words, we keep the original structure and dimension of `[blocks, instructions, tokens]` as input to preserve the execution logic and carry better semantics. In the past, it was very difficult to model the data structure this way, due to the padding issue where some instructions have a significantly larger number of subwords. Our implementation uses the recent ragged tensor interface in TensorFlow, which allows variable sequence length over the subword dimension without padding. Besides masking, we add a `CLS` token at the start of the assembly instructions, which acts as a condensed vector for a collective representation. `CLS` is left out for masking.

3.2 Code Fragment Encoding

Next, we obtain subword embeddings and aggregate them into instruction-level representation, before feeding it into the Transformer. Given a subword t of instruction, we map it to its subword embedding, and then couple it with a position embedding. After, we aggregate the position-encoded subword embedding into instruction-level embedding as e. Specifically:

$$e = 1/m * \sum_t \text{EM}(t) + \text{PE}_1(t) \tag{2}$$

where m is the instruction's subword count. The subword-level positional embedding is important in our case, since long strings and constants are broken down into different subwords. For example, the two constants 00f1 and f100 may yield the same set of subwords f1 and 00. With position encoding, the information order can be preserved. After obtaining the embedding for instructions, we feed them into a Transformer model. Suppose the embedding of all the instructions is denoted as E, and e is one of them, we have:

$$H = \text{TE}(E) \tag{3}$$

Note that in this step, the `CLS` token is treated as an individual instruction to distinguish it from the other instructions, and the Transformer only runs over the instruction-level embeddings. This design also helps us mitigate the sequence length limit of assembly code. Assembly code fragments are typically much longer than a natural language sentence in terms of subwords count. Now the limit applies to the number of instructions rather than the number of subwords.

3.3 Skip-gram Reconstruction Loss

In previous BERT-based methods such as Yu, et al. [10], the MLM task is to predict the masked tokens with the vector outputs from the Transformer corresponding to the time dimension:

$$y_j^{mlm} = \Theta(\boldsymbol{h}_j^i), \forall j \in M \tag{4}$$

where M is the masked tokens in an assembly instruction and Θ denotes a standard softmax function. The authors also propose other binary classification tasks including next sentence prediction (NSP), same graph prediction task (SGP), and graph classification task (GC). They use the first vector output from the Transformer for classification:

$$y_{nsp}, y_{sgp}, y_{gc} = \sigma(\boldsymbol{h}_0^i) \tag{5}$$

where σ denotes a sigmoid function. Although this approach is able to predict the masked token with full exposure to the token embedding, the vector used for the downstream task carries less information.

Similar to the MLM task in BERT, we have replaced the original tokens with a `MASK` token, swapped it with a random token from the corpus or kept it unchanged. Since we do not fully adopt BERT and only train the MLM task, the blocks are not sampled to be pairs, but individually fed into the network. The training task is to recover the masked or swapped tokens within each instruction using the output from the positional encoding layer. With all the instruction vectors h^i from the Transformer encoder,

we take the first one $\boldsymbol{h}^{cls}$, which is the CLS vector. Using a skip-gram approach, $\boldsymbol{h}^{cls}$ is treated as the paragraph (collectively representing all instructions in our case) representation. In order to model the position information, we duplicate the $\boldsymbol{h}^{cls}$ vector and tile it to the original instruction length L_{ins} to get the matrix h^{cls}. It is concatenated to another positional encoding layer PE_2 before the final prediction layer. PE_2 encodes the prediction position, which contains the mask locations. This way, we enable $\boldsymbol{h}^{cls}$ to also carry the position information while maintaining a condense learning representation. In particular:

$$\begin{aligned} p &= PE_2(\text{mask}) \\ h^p &= h^{cls} \oplus p \end{aligned} \tag{6}$$

where h^p is the final prediction vector, encoded with position corresponding to each masked instruction position. Each $\boldsymbol{h}_l^p$ predicts all m tokens within the instruction l. Formally, we define our MLM task to be:

$$y_1^l, ...y_m^l = \Phi(\boldsymbol{h}_l^p), \forall l \in L_{ins} \tag{7}$$

Φ denotes the sampled softmax [28], a more efficient method compared to the standard softmamx. For multiclass training, we need to train a function $F(x, y)$, which is GenTAL in this case, to compute the logits, which is the relative log probabilities of class y given the context x:

$$F(x_i, y) \leftarrow log(P(y|x)) + K(x) \tag{8}$$

where $K(x)$ is an arbitrary function that does not depend on y. We use a larger vocabulary size compared to NLP tasks, such as BERT, because of OOV and the training can slow down significantly. Sampled softmax can accelerate the process by reducing the softmax logits calculated for each training example by picking a small set of sampled classes $S \subset L$ using a sampling function $Q(y|x)$:

$$P(S_i = S|x_i) = \prod_{y \in S} Q(y|x_i) \prod_{y \in (L-S)} (1 - Q(y|x_i)) \tag{9}$$

Then a set of candidates $C_i = S_i \cup t_i$ can be created which contains the union of the target class and sampled classes. Through computing posterior probability that y is the target class given x_i and C_i, which is $P(t_i = y|x_i, C_i)$, we can obtain:

$$log(P(t_i = y|x_i, C_i)) = log(P(y|x_i)) - log(Q(y|x_i)) + K'(x_i, C_i) \tag{10}$$

which are the sampled relative logits that can feed into a softmax classifier. We can further write it as:

$$logits = F(x, y) - log(Q(x|y)) \tag{11}$$

4 Experiments

Experiments are conducted on a Linux server with 4 Nvidia RTX6000s, an Intel Xeon Gold 5218 CPU, and 300 GB of memory. The software used includes Python 3.8.6 and Tensorflow 2.4.1.

We collect two independent datasets, where one is for unsupervised training and the other is for evaluation. For the training set, we collect malware from MalwareBazaar and Malpedia[1] as well as benign system and software programs for Linux and Windows (see Table. 1). Each method is trained on 100,000 code fragments and roughly 1 million assembly instructions. The training set is split into 90% training and 10% validation for hyperparameter tuning. GenTAL is trained multiple times with different random seeds to estimate the variance. The vocabulary size is 80,000 for training the unigram tokenizer through the SentencePiece package. The maximum sequence length is 512, the embedding dimension is 192, the number of attention heads is 6, and the feedforward dimension is 32.

The evaluation dataset is generated by compiling four widely used utility and numeric libraries in different configurations. In practical scenarios, the ground truth similarity is difficult to obtain, especially on the block-level mapping. We present the first verified block-level assembly fragment mapping dataset. Matching on block-level is much difficult than on function-level, since it lacks many details present in the other blocks. Additionally, the function-level mapping used in the other related works cannot guarantee that the assembly code is actually mapping to the same source code, since optimization or obfuscation will mix source code from different functions and break function integrity. We generate mapping pairs based on (1) *cross-compiler* mapping blocks compiled with GCC and Clang, (2) *cross-optimization* mapping blocks with different optimizations, and (3) *cross-obfuscation* mapping original and obfuscated blocks [29].

[1] `https://malpedia.caad.fkie.fraunhofer.de/`, and `https://bazaar.abuse.ch/`

Table 1: Datasets Statistics

Training			Evaluation			
MalwareBazaar	Malpedia	Benign	OpenSSL	GMP	libtomcrypt	ImageMagick
86,225	3,158	343,235	4,777	1,351	590	2,733

4.1 COMPARED METHODS AND EVALUATION METRICS

We use mean rank (MRR) and precision at 1 (P@1) as the evaluation metrics, since the similarity detection task can be regarded as an information retrieval task. The following list contains the baselines we implement to compare against GenTAL:

TFIDF-str: TFIDF-based matching [30] by treating code as text. IDF is directly estimated on the testing set, so this approach has more edge over other DL-based methods.

TFIDF-mne: same as TFIDF-str, but we only use the operands of the instruction [30]. IDF is directly estimated on the testing set, so this approach has more edge over other DL-based methods.

MLM: BERT-based solution [10] that shows superior results compared to other binary code similarity detection works, such as Gemini [8] and Skip-thought [31], by a large margin, as well as other machine learning solutions. Only the MLM task is included in this variant.

MLM+NSP: Same as MLM, but with the additional NSP task.

MLM+SGP: Same as MLM, but this variant adds the same graph prediction (SGP) task to classify whether the pairs belong to the same graph.

MLM+SFP: Same as MLM, but we introduce an additional task in which to identify if the pairs are extracted from the same function to further add in more supervision.

MLM+GC: Same as MLM, but with the GC task (graph supervision) from Yu, et al [10]. The task includes classification of optimization level and architecture.

MLM+DIS: We implemented ELECTRA [21], where a discriminator is added to MLM to predict which tokens are originally masked, after the generator part of the model has recovered the tokens.

All-no-GC: This variant includes MLM, NSP, SGP, SFP, and DIS, except GC. GC includes supervision and can potentially correlate to the evaluation task.

ALL: We include all previous BERT-based tasks including GC.

4.2 RESULTS

We first evaluate the cross-compiler task between Clang and GCC shown in Table 2. The table on the top shows the evaluation when only the code fragments with the same number of instructions are used. The assembly code will be more similar than in the other situations, such as obfuscation. Therefore all baselines have relatively good performance. GenTAL is able to outperform most other baselines, except a slightly lower MRR for libtomcrypt against MLM+DIS. The evaluation of cross-compiler with different lengths is at the bottom. It is clear that similar pieces of binary code with different lengths can be much harder to detect, due to changed sequences and tokens. Moreover, when using O0, which is unoptimized, the outcomes between different compilers can vary more. We show that BERT-based models struggle to capture the semantics, regardless of the tasks learned, while GenTAL is able to significantly perform better under this circumstance. For the cross-optimization level evaluation shown in Table 3, O2-O3 is shown at the top and O0-O3 is at the bottom. For O2-O3, all methods have again good performance, since O3 is closer to O2 and both optimization levels usually perform the same. Note that all BERT-based models are nearly identical in this task, leading to another confirmation that additional tasks have little contribution to the specific downstream task. For O0-O3, there are gaps in the results, as shown in the bottom table. While GenTAL is able to achieve greater than 0.5 MRR most of the time, BERT-based models struggle to identify similar binary code, due to the larger difference between the two optimization levels. Finally, Table 4 shows the evaluation for the obfuscation task, where GenTAL leads all the other methods. Code obfuscation

Table 2: Cross-compiler evaluation: Clang O0 vs GCC O0

Same Length	OpenSSL MRR	OpenSSL P@1	GMP MRR	GMP P@1	libtomcrypt MRR	libtomcrypt P@1	ImageMagick MRR	ImageMagick P@1	Average MRR	Average P@1
TFIDF-mne [30]	0.46	0.38	0.72	0.60	0.66	0.50	0.31	0.21	0.54	0.42
TFIDF-str [30]	0.75	0.68	0.88	0.85	0.50	0.25	0.47	0.35	0.65	0.53
MLM [10]	0.61	0.55	0.72	0.60	0.68	0.50	0.38	0.28	0.60	0.48
MLM+NSP [10]	0.63	0.57	0.80	0.70	0.65	0.50	0.39	0.27	0.62	0.51
MLM+SGP [10]	0.62	0.56	0.78	0.65	0.61	0.44	0.38	0.26	0.60	0.48
MLM+SFP	0.62	0.56	0.81	0.70	0.61	0.44	0.38	0.26	0.61	0.49
MLM+GC [10]	0.62	0.55	0.80	0.70	0.59	0.38	0.38	0.28	0.60	0.48
MLM+DIS	0.65	0.58	0.76	0.65	0.63	0.38	0.45	0.34	0.63	0.49
All-no-GC	0.64	0.57	0.79	0.65	0.48	0.25	0.36	0.22	0.57	0.42
ALL [10]	0.66	0.58	0.85	0.75	0.69	0.56	0.44	0.32	0.66	0.55
GenTAL	**0.76**	**0.69**	**0.97**	**0.95**	**0.76**	**0.69**	**0.57**	**0.47**	**0.76**	**0.70**
Different Length	**OpenSSL MRR**	**OpenSSL P@1**	**GMP MRR**	**GMP P@1**	**libtomcrypt MRR**	**libtomcrypt P@1**	**ImageMagick MRR**	**ImageMagick P@1**	**Average MRR**	**Average P@1**
TFIDF-mne [30]	0.12	0.07	0.46	0.31	0.33	0.21	0.14	0.09	0.26	0.17
TFIDF-str [30]	0.35	0.29	0.57	0.47	0.39	0.27	0.25	0.19	0.39	0.30
MLM [10]	0.15	0.10	0.37	0.24	0.26	0.14	0.15	0.09	0.23	0.14
MLM+NSP [10]	0.16	0.11	0.40	0.26	0.27	0.15	0.17	0.10	0.25	0.16
MLM+SGP [10]	0.16	0.11	0.41	0.28	0.27	0.15	0.17	0.09	0.25	0.16
MLM+SFP	0.16	0.11	0.41	0.28	0.27	0.14	0.17	0.09	0.25	0.16
MLM+GC [10]	0.15	0.10	0.40	0.29	0.26	0.15	0.16	0.09	0.24	0.16
MLM+DIS	0.22	0.16	0.50	0.39	0.31	0.18	0.21	0.13	0.31	0.22
All-no-GC	0.17	0.11	0.44	0.32	0.27	0.13	0.16	0.09	0.26	0.16
ALL [10]	0.25	0.18	0.55	0.46	0.36	0.23	0.21	0.13	0.34	0.25
GenTAL	**0.40**	**0.33**	**0.69**	**0.61**	**0.49**	**0.36**	**0.35**	**0.28**	**0.48**	**0.39**

Table 3: Cross-optimimzation level evaluation O2 vs O3 O0 vs O3

O2-O3	OpenSSL MRR	OpenSSL P@1	GMP MRR	GMP P@1	libtomcrypt MRR	libtomcrypt P@1	ImageMagick MRR	ImageMagick P@1	Average MRR	Average P@1
TFIDF-mne [30]	0.20	0.13	0.83	0.75	1.00	1.00	0.47	0.34	0.63	0.56
TFIDF-str [30]	0.82	0.80	1.00	1.00	0.96	0.93	0.93	0.91	0.93	0.91
MLM [10]	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.90	0.93	0.91
MLM+NSP [10]	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.90	0.93	0.91
MLM+SGP [10]	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.90	0.93	0.91
MLM+SFP	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.90	0.93	0.91
MLM+GC [10]	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.91	0.93	0.91
MLM+DIS	0.82	0.81	1.00	1.00	0.96	0.93	0.91	0.89	0.93	0.91
All-no-GC	0.82	0.81	1.00	1.00	1.00	1.00	0.90	0.88	0.93	0.92
ALL [10]	0.82	0.81	1.00	1.00	0.96	0.93	0.92	0.90	0.93	0.91
GenTAL	**0.87**	**0.83**	**1.00**	**1.00**	**1.00**	**1.00**	**0.94**	**0.91**	**0.95**	**0.94**
O0-O3	**OpenSSL MRR**	**OpenSSL P@1**	**GMP MRR**	**GMP P@1**	**libtomcrypt MRR**	**libtomcrypt P@1**	**ImageMagick MRR**	**ImageMagick P@1**	**Average MRR**	**Average P@1**
TFIDF-mne [30]	0.05	0.02	0.35	0.23	0.52	0.36	0.11	0.06	0.26	0.17
TFIDF-str [30]	**0.48**	**0.35**	0.62	0.52	0.68	0.57	**0.73**	**0.66**	0.63	0.53
MLM [10]	0.08	0.05	0.28	0.19	0.56	0.36	0.25	0.18	0.29	0.19
MLM+NSP [10]	0.08	0.05	0.26	0.16	0.53	0.29	0.22	0.14	0.27	0.16
MLM+SGP [10]	0.08	0.04	0.26	0.15	0.56	0.36	0.22	0.13	0.28	0.17
MLM+SFP	0.07	0.04	0.25	0.15	0.56	0.36	0.21	0.13	0.28	0.17
MLM+GC [10]	0.08	0.04	0.26	0.16	0.53	0.29	0.21	0.15	0.27	0.16
MLM+DIS	0.15	0.08	0.36	0.24	0.67	0.50	0.30	0.21	0.37	0.26
All-no-GC	0.06	0.03	0.25	0.14	0.52	0.36	0.23	0.16	0.27	0.17
ALL [10]	0.14	0.07	0.41	0.27	0.67	0.50	0.23	0.15	0.36	0.25
GenTAL	0.46	0.34	**0.78**	**0.67**	**0.70**	**0.57**	0.67	0.61	**0.65**	**0.55**

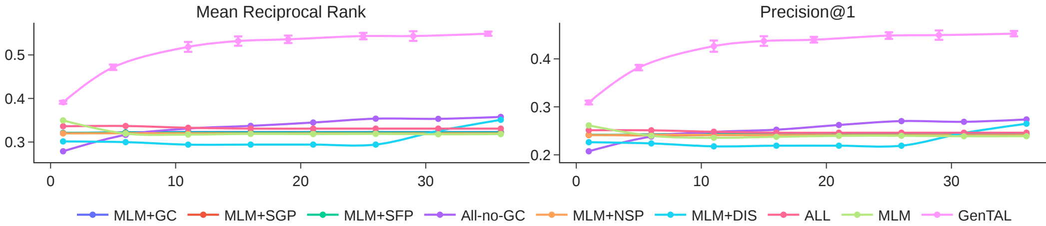

Figure 3: Performance on unseen obfuscated code over training epochs. GenTAL's error bars are also shown.

Table 4: Obfuscation evaluation with GCC O0

	OpenSSL		GMP		libtomcrypt		ImageMagick		Average	
	MRR	P@1	MRR	P@1	MRR	P@1	MRR	P@1	MRR	P@1
TFIDF-mne [30]	0.51	0.41	**0.46**	**0.33**	0.36	0.26	0.42	0.31	0.44	0.33
TFIDF-str [30]	0.67	0.59	0.25	0.19	0.58	0.45	0.42	0.33	0.48	0.39
MLM [10]	0.48	0.39	0.14	0.09	0.34	0.23	0.23	0.16	0.30	0.22
MLM+NSP [10]	0.51	0.43	0.16	0.10	0.36	0.25	0.25	0.18	0.32	0.24
MLM+SGP [10]	0.51	0.43	0.15	0.10	0.37	0.26	0.25	0.18	0.32	0.24
MLM+SFP	0.51	0.43	0.16	0.10	0.37	0.26	0.25	0.18	0.32	0.24
MLM+GC [10]	0.53	0.44	0.17	0.11	0.37	0.26	0.27	0.20	0.34	0.25
MLM+DIS	0.53	0.43	0.19	0.13	0.39	0.28	0.31	0.23	0.35	0.27
All-no-GC	0.49	0.40	0.17	0.12	0.36	0.25	0.26	0.19	0.32	0.24
ALL [10]	0.56	0.47	0.20	0.14	0.39	0.28	0.31	0.23	0.36	0.28
GenTAL	**0.74**	**0.64**	0.40	0.31	**0.57**	**0.46**	**0.51**	**0.40**	**0.55**	**0.45**

is the most impactful similarity detection task, since malware often bypass detection software using it.

Overall, we can show that GenTAL consistently outperforms all baselines under all compiling configurations, with significant margins. We observe that for more difficult tasks, such as cross-compiler with different lengths, O0-O3, and obfuscation, our design still has reasonable performance with the MRR ranging from 0.3 to 0.7. GenTAL only trains with the MLM task. Therefore it also eliminates the need for sampling pairs for training additional tasks, such as NSP and SFP. Moreover, the model parameter is much smaller compared to the BERT-based models, since we use only 1 Transformer encoder and much smaller dimensions in the other layers. The additional training tasks on top of MLM result in little to no performance gain as shown in all tables. In Fig. 3, we plot the training epochs vs. similarity detection evaluation on the fly for the obfuscation task. Even with only learning the MLM task, GenTAL's evaluation performance increases faster than for all the other baselines. Therefore, we believe that a more condensed vector using skip-gram contains more semantic information and can result in better similarity detection.

5 CONCLUSION

In this paper, we propose GenTAL, a generative approach with denoising skip-gram and Transformer for binary code similarity detection. We improve upon the BERT-based representation learning specifically for assembly code, by introducing the PVDM paradigm to condense the code fragment representation. This design allows us to capture more semantics from assembly code, which is important as often the compiled code can have different lengths and tokens based on configurations, such as optimization levels, compilers, and obfuscations. When evaluating GenTAL, we show that its performance is superior to both BERT-based and TFIDF-based methods for all the different compiler configurations. In practice, our approach is able to train with fewer data and adapt to more scenarios due to its generative nature. The limitation of GenTAL is that the downstream similarity detection performance can be hindered by the OOV problem, especially for heavy numeric libraries with many constants. Although GenTAL achieves better generalizability than the state-of-the-art solutions, there are still areas or tasks, such as other obfuscation methods, which were not evaluated.

REFERENCES

[1] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 2006.

[2] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[3] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "BinSign: Fingerprinting binary functions to support automated analysis of code executables," in *Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017.

[4] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and robust binary library function identification using function shape," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[5] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proc. of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. San Francisco, CA: ACM Press, August 2016, pp. 461–470.

[6] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[7] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension*, 2017.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

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

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

[16] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, "Bindeep: A deep learning approach to binary code similarity detection," *Expert Systems with Applications*, vol. 168, p. 114348, 2021.

[17] 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.

[18] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *Sstic*, vol. 5, no. 1, p. 3, 2005.

[19] M. Bourquin, A. King, and E. Robbins, “Binslayer: accurate comparison of binary executables,” in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013, pp. 1–10.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *arXiv preprint arXiv:1706.03762*, 2017.

[21] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, “Electra: Pre-training text encoders as discriminators rather than generators,” *arXiv preprint arXiv:2003.10555*, 2020.

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

[23] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[24] P. Gage, “A new algorithm for data compression,” *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[25] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[26] T. Kudo, “Subword regularization: Improving neural network translation models with multiple subword candidates,” *arXiv preprint arXiv:1804.10959*, 2018.

[27] T. Kudo and J. Richardson, “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing,” *arXiv preprint arXiv:1808.06226*, 2018.

[28] S. Jean, K. Cho, R. Memisevic, and Y. Bengio, “On using very large target vocabulary for neural machine translation,” *arXiv preprint arXiv:1412.2007*, 2014.

[29] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-llvm–software protection for the masses,” in *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 3–9.

[30] W. M. Khoo, A. Mycroft, and R. Anderson, “Rendezvous: A search engine for binary code,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 329–338.

[31] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler, “Skip-thought vectors,” *arXiv preprint arXiv:1506.06726*, 2015.