Highlights

SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings
Sun Ruijin, Guo Shize, Guo Jinhong, Li Wei, Zhan Dazhi, Guo Xi, Sun Meng, Pan Zhisong

- To investigate the robustness of the embeddings, our finding reveals that SOTA models encounter expression degradation issues, and the reasons for these problems are analyzed.
- This is the first article that employs unsupervised contrastive learning in Binary Code Similarity Detection. Our model effectively circumvents the labelling difficulties in existing approaches.
- Our findings indicate that SimCLF outperforms existing models. Additionally, SimCLF is tested on a real-world vulnerability dataset and it has achieved flawless results.
- SimCLF demonstrates greater efficiency in few-shot settings.
- We hope that our models will inspire others. We have released the code of SimCLF publicly available at https://github.com/iamawhalez/fun2vec.
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

Sun Ruijin\textsuperscript{a}, Guo Shize\textsuperscript{b}, Guo Jinhong\textsuperscript{c}, Li Wei\textsuperscript{a}, Zhan Dazhi\textsuperscript{a}, Guo Xi\textsuperscript{d}, Sun Meng\textsuperscript{a} and Pan Zhisong\textsuperscript{a,*}

\textsuperscript{a}Army Engineering University of PLA, Nanjing, 210007, China
\textsuperscript{b}National Computer Network and Information Security Management Center, Beijing, 100029, China
\textsuperscript{c}Shanghai Jiao Tong University, Shanghai, 200240, China
\textsuperscript{d}University Of Science & Technology Beijing, Beijing, 100083, China

ARTICLE INFO

Keywords:
Binary code similarity detection
Unsupervised contrastive learning
Embedding learning
Cyberspace security

ABSTRACT

Function-level binary code similarity detection is a crucial aspect of cybersecurity. It enables the detection of bugs and patent infringements in released software and plays a pivotal role in preventing supply chain attacks. A practical embedding learning framework relies on the robustness of the assembly code representation and the accuracy of function-pair annotation, which is traditionally accomplished using supervised learning-based frameworks. However, annotating different function pairs with accurate labels poses considerable challenges. These supervised learning methods can be easily overtrained and suffer from representation robustness problems. To address these challenges, we propose SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings. We take an unsupervised learning approach and formulate binary code similarity detection as instance discrimination. SimCLF directly operates on disassembled binary functions and could be implemented with any encoder. It does not require manually annotated information but only augmented data. Augmented data is generated using compiler optimization options and code obfuscation techniques. The experimental results demonstrate that SimCLF surpasses the state-of-the-art in accuracy and has a significant advantage in few-shot settings.

1. Introduction

Function-level binary code similarity detection (BCSD) is used to analyze whether two given binary code snippets are similar in semantics. It plays an important role in various tasks of cyberspace security [18], such as bug search [29], plagiarism detection [31], vulnerability discovery [14], malware detection [22], and malware clustering [21]. Several researchers [35][28][32][25][45] have demonstrated promising results in the analysis of binary code similarity. There are many challenges associated with BCSD. The foremost one is the huge number of the functions to be detected. As the Internet era progresses, new functions and new applications emerge in an endless stream, and the number of binary files increases rapidly [32]. In addition to the huge number, the architectures of the devices are becoming more and more diverse. Besides the traditional x86 with Windows, there are a variety of other operating systems (Ubuntu, IOS, Android etc.) and CPU types (Arm, MIPS etc.). When new software are developed, people often do not start from scratch. They may reuse existing code snippets or open source libraries. If there is a bug in the original library file, that bug will be present in all programs that reuse it. It is impossible to manually identify all the library files that software contains in the face of massive data. Therefore, there is an urgent need for automated methods based on the semantics of binary files.

Despite the highly practical and academic value of BCSD, progress has been very slow because of the lack of quality labelled datasets in the BCSD field. Annotating binary function pairs is a challenging task, and security analysts often face difficulty in obtaining the source code of the software being analyzed for various reasons. Additionally, compiled binary files do not contain high-level language components such as variable names and data structures, as most of the program semantics are lost during the compilation process. Binaries generated from the same high-level language can undergo significant changes when compiled using different compilers, optimization options, and

ORCID(s):
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

Figure 1: The figure displays two-dimensional visualizations of embeddings. Figure (a) shows the visualization of embeddings generated from a pre-trained transformer model called TREX [35]. On the other hand, Figure (b) shows the visualization of embeddings trained using Word2Vec [34].

on different CPU architectures. Additionally, hackers often use obfuscation techniques to distort binary code, making BCSD an increasingly difficult task. Unlike labelling in Natural Language Processing (NLP), labelling binary code requires many specialists and a lot of hard work. All of these issues contribute to difficulties in annotating binary code. Despite overcoming the aforementioned challenges, labelling different function pairs accurately remains challenging. Supervised learning-based methods typically use distance functions, such as edit distance and cosine similarity [32], to label function pairs. The distance between the two similar functions could be labelled as \{-1\} and the two different functions as \{-1\} or \{0\}. But the mutual information between any pair of the function could not be either \{-1\} or \{0\}. For example, we may label \textit{Recv()} and \textit{RecvAndWrite()} as \{-1\}, because they are not the same function. However, in reality, these two functions share significant similarities, the ground truth should be a score between \{-1\} and \{1\}. Finding an appropriate number for labelling different function pairs can be challenging. If we use \{-1\} to label them, it can lead to overtraining which causes a slow convergence.

A reliable framework for binary code similarity analysis depends on a robust vector representation of assembly code. While current methods have achieved state-of-the-art (SOTA) results in BCSD [35][28][48][25], many pretrained models face the issue of representation degeneration. To address this, Singular Value Decomposition (SVD) is utilized to approximate the learned embeddings of the SOTA model TREX [35]. As shown in Figure 1, the rank-2 approximation of the embeddings for all datasets that used in the finetuning process is plotted. The detail of the dataset is described in Section 4.1. These embeddings are compared with those trained by Word2Vec [34]. The 2D visualization of the representations from the pre-trained model collapses to a corner, while the word embeddings learned from Word2Vec [34] are uniformly distributed.

Motivated by the latest developments in the area of contrastive learning [44][16][5][47], we propose a novel binary code semantic difference (BCSD) model at the function level, namely SimCLF. SimCLF is a self-supervised framework that leverages contrastive learning and solely relies on positive examples and sample augmentation techniques. This approach helps to avoid the over-training problem that may arise from the inaccurate labelling of negative pairs. Additionally, the experimental findings demonstrate that SimCLF is capable of eliminating the anisotropy [12] caused by the pre-trained model. In terms of accuracy, it surpasses existing models such as TREX [35] and jTrans [42]. In summary, our contributions are as follows:

- To investigate the robustness of the embeddings generated by SOTA pre-training models, several sets of metrics are developed. The findings suggest that these models suffer from expression degradation issues, and the underlying causes of these problems have been analyzed.

- We propose SimCLF, a simple contrastive learning framework of function-level binary embeddings. As far as we know, this is the first article that utilizes the unsupervised contrastive learning model in BCSD. SimCLF avoids the labelling difficulties that arise in supervised models.
• The model is subjected to rigorous evaluation across multiple datasets, and its performance is compared against the existing SOTA approaches. The results reveal that SimCLF surpasses the performance of its counterparts. Furthermore, SimCLF is applied to a real-world vulnerability dataset, where it has achieved perfect results.

• Our model is more efficient in few-shot settings, as it achieves better performance than TREX [35] with only 2,048 binary pairs. In comparison, TREX requires about 89,809 labelled binary pairs.

• It is hoped that SimCLF will inspire others in the field. To facilitate further research, the code for SimCLF have been released at https://github.com/iamawhalez/fun2vec.

2. Background and Related Work

In this section, we provide a brief introduction to the definition of BCSD and related work in the field.

2.1. Definition of BCSD

The vast majority of software development is characterized by the practice of code reuse, it is rare for developers to initiate a project from scratch. As a result, a significant number of code clones are likely to exist within the underlying assembly code. An efficient BCSD engine can greatly reduce the burden of the manual analysis process involved in reverse engineering. By exploiting the large amount of available binary data, BCSD can cater to the information requirements of downstream tasks. The definition of BCSD can be described as follows: given two pieces of binary code, without any supplementary information such as source code or compilation details, the task is to determine whether the two pieces of code are similar. In this context, similarity refers to semantic consistency, but not necessarily the same characters. In other words, if similar binary functions belong to different architectures or are compiled using different optimizations, they will still behave the same way when they are provided with the same input. BCSD methods can be broadly categorized into two distinct groups: feature-based methods and embedding-based methods.

2.2. BCSD Based on Features

Traditional BCSD methods heavily rely on specific features or structures. The earliest tool for comparing binary file similarity, named EXEDIFF [3], was released in 1999 and was used to determine the patch information of different versions of executable files. These methods primarily relied on disassembled code structure information, and string-based detection techniques are the most fundamental methods [2][4]. String-based techniques use string or string edit distance to detect duplicates. Another commonly used feature is \( n \)-\textit{grams} wherein sliding windows of size \( n \) in bytes are considered as content [24]. Many articles adopted a graph-based approach for BCSD. Graph-based methods can be broadly categorized into those that utilize Program Dependency Graphs (PDGs) and those that use Control Flow Graphs (CFGs). A PDG is a graphical representation of the data and control dependencies within a procedure [30]. It was used in program slicing at first [46]. More people use CFG (Control Flow Graph), such as [36] and [37]. These methods transform binary code into CFGs and use graph analysis algorithms to determine similarity.

2.3. BCSD Based on Embeddings

Embedding is the transformation of discrete variables into a high-dimensional continuous space. In embedding-based solutions, functions are not compared based on the function itself, but instead on the learned embedding of functions. First, binary functions are transformed into multi-dimensional vector representations (embeddings). These embeddings are then compared using simple geometric operations [32]. Functions with similar semantics have embeddings that are closer to each other in the high-dimensional space.

In literature [11], the authors proposed a function embedding solution called Asm2Vec. Asm2Vec transforms the assembly code into embeddings by Word2Vec [34]. The function representations are learned by using code embeddings based on the PV-DM model [26]. Some articles employ graph embedding methods. For instance, [45] use a neural network-based approach to learn the embedding of an Attributed Control Flow Graph (ACFG). Initially, SAFE [32] employs Word2Vec to convert the instructions into embeddings. A GRU Recurrent Neural Network (GRU RNN) is then used to capture the sequential interaction of the instructions.

The success of pure attention networks in the NLP domain such as Transformer [40] and BERT [10] has inspired the adoption of Transformer-like structures in BCSD. All of these articles [25][28][35][48] employ architectural structures that are rooted in the pure attention pre-training model, yielding remarkable outcomes. In [48], the authors adopt a method based on BERT, where they pre-train the binary code using BERT at both the one-token and one-block
levels. Then Convolutional Neural Networks (CNN) are used on adjacency matrices to extract the order information. PalmTree [28] utilizes three pre-training tasks to capture various characteristics of assembly language and generate general-purpose instruction embeddings. These pre-training tasks are performed through self-supervised training on large-scale unlabelled binary corpora. TREX [35] develops a novel neural architecture, a hierarchical Transformer, which can learn execution semantics from micro-traces during the pre-training phase. jTrans[42] is the first solution that embeds control flow information of binary code into Transformer-based language models. However, none of these papers have investigated the quality of the embedding that they’ve learned. Our experiments indicated that the issue of representation collapse exists in code embedding based on pre-trained models.

3. The Basic Structure of the Model

In this section, the basic structure of SimCLF model is presented, and the technical approach behind it is discussed. We start by describing the general algorithm, then briefly introduce the major parts of the model. Throughout this paper, the following mathematical notations are adopted. The element or binary function in the original dataset \( X \) is denoted as \( x \), while \( f(\cdot) \) represents a deep neural network-based encoder. We primarily use BERT-like pretrained model TREX [35] and jTrans[42] as the encoder. We aim at fine-tuning the encoder \( f(\cdot) \) to make the function representation more robust. \( \text{Trans}(\cdot) \) is the data augmentation function. \( z \) is the learned embedding of the function \( x \). \( \tau \) is a temperature parameter. \( d(\cdot) \) is a distance function, \( \mathcal{L}(\cdot) \) is a loss function.

3.1. General Framework

Inspired by the advancements in contrastive learning and natural language processing, we propose SimCLF, which aims to address the two challenges posed in the Section 1. The model consists of a siamese network, which implements the proposed encoder, and a contrastive learning model as shown in Figure 2. By employing the unsupervised approach, the manual labor involved in labelling different function pairs can be circumvented, while simultaneously enhancing the quality of the embedding to a significant extent. SimCLF consists of three parts.

- A data set \( X \) and the sample augmentation module \( \text{Trans}(\cdot) \).
- A DNN-based encoder \( f(\cdot) \) that projects input function pairs into low-dimensional.
- A contrastive learning module that is used to optimize the encoder, which consists of a distance function \( d(\cdot) \) and a loss function \( \mathcal{L}(\cdot) \).

![Figure 2: The proposed unsupervised learning method with siamese networks follows a framework where input function pairs are projected into low-dimensional, normalized embeddings through the use of a DNN-based encoder. After undergoing the fine-tuning process, the feature representations of the same function instance with different data augmentations are invariant, while the embeddings of different function instances are distributed.](image-url)

Given a function \( x_i \) in the original dataset \( X \), the augmented data are produced by passing the original \( x_i \) to the data augmentation module \( \text{Trans}(\cdot) \). Two augmented binary functions, \( x^1_i \) and \( x^2_i \), are generated. Then, the two embeddings \( z^1_i \) and \( z^2_i \) of the function-pair are produced. \( z^1_i = f(x^1_i) \) and \( z^2_i = f(x^2_i) \) where \( f \) is the representation learning module, are also known as the encoder. After generating augmented data through the data augmentation module, the contrastive learning module is used to optimize the encoder. A key concept introduced here is non-parametric instance discrimination [44]. We treat each function instance as a distinct class of its own and train a classifier to discriminate
between individual instance classes. In other words, given any function pair \( x_i, x_j \), if \( x_i \) and \( x_j \) are semantically different, it is believed that they are members of different classes. The distance function \( d(i, j) \) between the embeddings of \( x_i \) and \( x_j \) should be greater than a threshold value. If \( x_i \) and \( x_j \) are semantically the same, it is believed that they belong to the same classes. And the distance function \( d(i, j) \) between the embeddings of \( x_i \) and \( x_j \) should be less than a threshold value.

### 3.2. Data Augmentation Strategies

The data augmentation module plays a crucial role in contrastive learning. In the realm of binary code, binary codes with the same meaning may have completely different assemblies on different devices. The data augmentation module is capable of generating a diversity of assemblies. Given a binary function code, different binaries are produced with the same semantics, which may be suitable for different architectures or compiled with different optimizations or obfuscation techniques. In this paper, three different data augmentation strategies are explored, namely cross-architecture, cross-optimization, and obfuscation, to generate augmented instances for the contrastive learning process.

**Cross-architecture.** In general, the CPU platforms are randomly selected on which the code will run. The two main parameters in this process are the CPU architecture and width. The architectures include x86-32, x86-64, arm-32, arm-64, mips-32, and mips-64, covering the most common mainstream CPU types.

**Cross-optimization.** The assembly codes produced by different compilation options for the same function written in the same high-level language are also very different. Given a specific function, it will be compiled into different architectures with 4 optimization options, i.e., O0, O1, O2 and O3 and 2 compilers GCC-5.4, GCC-7.5.

**Obfuscation.** Hackers often use code obfuscation techniques to prevent security personnel and tools from analyzing their code. To simulate this scenario, Hikari [50] is used to generate obfuscated augmented instances for the contrastive learning process. Several common obfuscation techniques are applied, including Pseudo Control Flow, Instruction Substitution, Basic Block Splitting, and others.

### 3.3. Function Representation Learning Module

In this section, the binary function representation learning module that we adopted are briefly introduced. Given a specific pair of binary functions, they are converted into continuous vectors in high dimensional space by the function representation learning module. The similarity of the binary pair is then calculated according to the distance of two vectors. SimCLF allows for any network architecture for encoding. Two model TREX[35] and jTrans [42] are chosen based on the following considerations. Firstly, these two model are the currently SOTA models. Pre-training based models are currently the most promising models, and many articles [35][28][48][25] have adopted it. Secondly, TREX and jTrans have open-sourced their codes, unlike some other models which are not fully disclosed. This allowed us to easily incorporate it into our framework and adapt it to suit our needs.

TREX is a pre-trained language model. It can generate embeddings for each token in an assembly function. TREX could encode the input functions and obtain their respective embeddings in the vector space. The function representations are obtained by performing average pooling of the token embeddings at the last layer.

jTrans is based on the BERT architecture. It capture control-flow information, by sharing parameters between token embeddings and position embeddings for each jump target of instructions. jTrans is trained by unsupervised learning tasks to learn the semantics of instructions and control-flow information of binary functions. Finally, the BertPooler layer is used to obtain the function embedding.

### 3.4. Contrastive Learning Module

In this section, a detailed description of the contrastive learning module is provided, which consists of two components: a distance function and a loss function. To evaluate the similarity of function pairs, the cosine distance function \( \text{sim}() \) are used. Given a pair of functions \( x_1 \) and \( x_2 \), \( z_1 = f(\vec{x}_1) \) and \( z_2 = f(\vec{x}_2) \) are the respective embedding vectors produced by the encoder. \( w \) is the dimension of embeddings. Cosine similarity are used as the distance metric \( d(\cdot) \), which is calculated using the following formula:

\[
d(z_1, z_2) = \text{sim}(f(x_1), f(x_2)) = \frac{\sum_{i=1}^{w} (f(\vec{x}_1)[i] \cdot f(\vec{x}_2)[i])}{\sqrt{\sum_{i=1}^{w} f(\vec{x}_1)[i]^2} \cdot \sqrt{\sum_{i=1}^{w} f(\vec{x}_2)[i]^2}}.
\]

---

Sun Ruijin et al: Preprint submitted to Elsevier
and \( f(x)[i] \) indicates the \( i \)-th component of the embedding vector of the function \( x \). The cosine distance between two function embeddings ranges from \([-1, 1]\), where a distance of 1 indicates that the functions are identical and a distance of -1 indicates that they are completely different.

If there are \( n \) different functions, \( f(x_i), i \in [1, n] \) is the learned representation of the \( i \)-th function. We enforce \( \|z\| = 1 \) via an \( L2\)-normalization layer. Following the approach described in reference [44], the probability are formulated that any embedding \( z \) belongs to the \( i \)-th function as an instance discrimination objective using the softmax-like criterion,

\[
P(z_i | z) = \frac{\exp(z^T z)}{\sum_{j=1}^{n} \exp(z^T z)},
\]

The distance function is substituted into equation 2, where \( \text{sim}(z_i, z) \) denotes the dot product between L2 normalized \( z_i \) and \( z \).

\[
P(z_i | z) = \frac{\exp(\text{sim}(z_i, z))}{\sum_{j=1}^{n} \exp(\text{sim}(z_i, z))},
\]

In the context of the given scenario, \( P(z_i | z) \) represents the probability that the embedding \( z \) belongs to class \( i \). A temperature parameter \( \tau \) is incorporated to regulate the concentration level [20]. The probability \( P(z_i | z) \) is modified as follows:

\[
P(z_i | z) = \frac{\exp(\text{sim}(z_i, z)/\tau)}{\sum_{j=1}^{n} \exp(\text{sim}(z_i, z)/\tau)},
\]

Our objective is to maximize the joint probability \( \prod_{i=1}^{n} P(z_i | z) \), which is equivalent to minimizing the negative log-likelihood \( -\sum_{i=1}^{n} \log P(z_i | z) \). The loss function becomes the NT-Xent (the normalized temperature-scaled cross-entropy loss) [5], which needs to be minimized during the fine-tuning process.

During each training epoch, a particular function \( x_i \) in the batch of \( N \) is selected. And two augmented functions \( x_i^1 \) and \( x_i^2 \) of the same semantic are randomly generated. This results in a total of \( 2N \) data points denoted as \( z_n, n \in [1, 2N] \).

In our approach, negative examples are not explicitly sampled. Instead, when given a positive pair \( x_i^1 \) and \( x_i^2 \), the corresponding vectors are \( z_{2i} = f(x_i^1) \) and \( z_{2i+1} = f(x_i^2) \). The remaining \( 2(N-1) \) augmented examples within the minibatch are treated as negative examples. The vector \( z_{2i} \) is similar to \( z_{2i+1} \), and distinct from all the other \( 2(N-1) \) augmented examples. Consequently, the loss function of \( x_i^1 \) and \( x_i^2 \) could be formulated as

\[
\mathcal{L}_{x_i^1} = -\log \frac{\exp(\text{sim}(z_{2i}, z_{2i+1})/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(z_{2i}, z_k)/\tau)}, \quad \mathcal{L}_{x_i^2} = -\log \frac{\exp(\text{sim}(z_{2i+1}, z_{2i})/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(z_{2i+1}, z_k)/\tau)}.
\]

Then the loss function for a positive pair of examples \( x_i^1 \) and \( x_i^2 \) is defined as:

\[
\mathcal{L}_{x_i} = 1/2(\mathcal{L}_{x_i^1} + \mathcal{L}_{x_i^2}) = 1/2 \left( \log \frac{\exp(\text{sim}(z_{2i}, z_{2i+1})/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(z_{2i}, z_k)/\tau)} + \log \frac{\exp(\text{sim}(z_{2i+1}, z_{2i})/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(z_{2i+1}, z_k)/\tau)} \right).
\]

The final loss \( \mathcal{L} \) is computed over all positive pairs, where \( \mathcal{L} = 1/N \sum_{i=1}^{N} \mathcal{L}_{x_i} \). The employed loss function \( \mathcal{L} \) effectively maximizes the dissimilarity between a function and its augmented data, while simultaneously minimizing the cosine distance between distinct functions. Through the minimization of this loss function, we are able to achieve instance classification without relying on labeled samples of different types. The complete algorithm outlining the entire process is presented in Algorithm 1.

**3.5. Why does SimCLF work**

SimCLF is the first paper to apply unsupervised contrastive learning in the field of BCSD. Unsupervised contrastive learning has garnered remarkable success, whereas the underlying mechanism of contrastive loss has received
More precisely, the gradients associated with the two negative pairs of the gradients pertaining to the SimCLF loss function, it has been discovered that SimCLF is sensitive to hard examples. Automatic selection of these hard examples can make training more effective and efficient [39]. Through an analysis of the underlying principles of SimCLF. First, SimCLF demonstrates sensitivity to hard examples comparatively less scholarly attention. Taking inspiration from [43][16][41], our objective is to offer a comprehensive investigation into the underlying principles of SimCLF. First, SimCLF demonstrates sensitivity to hard examples. Automatic selection of these hard examples can make training more effective and efficient [39]. Through an analysis of the gradients pertaining to the SimCLF loss function, it has been discovered that SimCLF is sensitive to hard examples. More precisely, the gradients associated with the two negative pairs m and n, where m, n ≠ 2i, 2i + 1 are formulated as:

$$\frac{\partial (L(x_i^1))}{\partial (\text{sim}(z_{2i}, z_m))} = \frac{1}{\tau} \frac{\exp(\text{sim}(z_{2i}, z_m)/\tau)}{\sum_{k=1}^{2N\mid \{k\neq 2i\}} \exp(\text{sim}(z_{2i}, z_k)/\tau)},$$

(7)

$$\frac{\partial L(x_i^1)}{\partial (\text{sim}(z_{2i}, z_n))} = \frac{1}{\tau} \frac{\exp(\text{sim}(z_{2i}, z_n)/\tau)}{\sum_{k=1}^{2N\mid \{k\neq 2i\}} \exp(\text{sim}(z_{2i}, z_k)/\tau)},$$

(8)

The observation that the gradients with respect to negative samples are proportional to the exponential term, it suggests that SimCLF exhibits sensitivity to hard examples.

$$\frac{\partial L(x_i^1)}{\partial (\text{sim}(z_{2i}, z_m))}/\frac{L(x_i^1)}{\partial (\text{sim}(z_{2i}, z_n))} = \frac{\exp(\text{sim}(z_{2i}, z_m)/\tau)}{\exp(\text{sim}(z_{2i}, z_n)/\tau)},$$

(9)

Second, SimCLF significantly diminishes both alignment and uniformity. In a recent study, Wang and Isola [43] identified two essential properties pertaining to contrastive learning, namely alignment and uniformity. These properties serve as metrics for evaluating the quality of learned representations. Let p_{pos}(·) denote the distribution of positive pairs. The concept of alignment aims to quantify the distance between embeddings of positive paired instances.

$$\text{Alignment} = \mathbb{E}_{x_i^1, x_i^2 \in p_{pos}(·)} \| f(x_i^1) - f(x_i^2) \|,$$

(10)

Let p_{data}(·) be the negative data distribution. Uniformity measures how well the embeddings of negative pairs are uniformly distributed.

$$U_{nf} = \mathbb{E}_{x, y \sim p_{data}} e^{-2\| f(x) - f(y) \|^2},$$

(11)
Let the loss function of $x_i^1$ could be formulated as

$$\mathcal{L}_{x_i^1} = -\log \frac{\exp(\text{sim}(z_2i, z_{2i+1})/\tau)}{\sum_{k=1, k \neq 2i}^{2N} \exp(\text{sim}(z_2i, z_k)/\tau)}.$$  \hfill (12)

It can be divided into two parts

$$\mathcal{L}_{x_i^1} = -\frac{1}{\tau} \sum \text{sim}(z_2i, z_{2i+1}) + \log \sum_{k=1, k \neq 2i}^{2N} \exp(\text{sim}(z_2i, z_k)/\tau).$$ \hfill (13)

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (\mathcal{L}_{x_i^1} + \mathcal{L}_{x_i^2}).$$

Because $\mathcal{L}_{x_i^1}$ and $\mathcal{L}_{x_i^2}$ is completely symmetrical, we take $\mathcal{L}_{x_i^1}$ as an example.

$$\frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{x_i^1} = \frac{1}{N} \left( -\frac{1}{\tau} \sum_{i=1}^{N} \text{sim}(z_2i, z_{2i+1}) + \sum_{i=1}^{N} \log \sum_{k=1, k \neq 2i}^{2N} \exp(\text{sim}(z_2i, z_k)/\tau) \right).$$ \hfill (14)

The first term of the loss function maintains the similarity of positive instances, while the second term acts to separate negative pairs. The loss function is mathematically expressed by the Equation (14). As indicated by [43], in the scenario where the number of instances tends towards infinity and $f(\cdot)$ is normalized, the first term is exactly the alignment term as shown in Equation (10). Regardless of the constant factors, the second term signifies favorable uniformity [16][23]. Thus, the optimization of the InfoNCE loss facilitates the attainment of both favorable alignment and favorable uniformity.

Third, the data augmentation module plays a crucial role in contrastive learning. Richer data augmentation indicates better downstream performance [23]. SimCLF encompasses an extensive data augmentation methods, including the most common mainstream CPU types, 4 optimization options and different compilers. The effectiveness of the model has been theoretically examined, with the rationale behind its efficacy discussed. To further substantiate its effectiveness, the key metrics will be validated in Section 4.

4. Experiment and Evaluation

The dataset and experimental configuration are introduced in this section. Our evaluation aims to answer the following question:

RQ1. Basic statistical laws of assembly code. What are the basic statistical laws of assembly code, and is there an expression degradation problem in the results of the original pre-trained model? Additionally, we investigate whether fine-tuning the model alleviates the expression degradation issue.

RQ2. Accuracy of the model. What is the accuracy of our contrastive learning model, and how does it compare to the accuracy of traditional SOTA models?

RQ3. Ablation Research. How do the different parts contribute to improving the performance of the model?

RQ4. Performance under few-shot Settings.

RQ5. Uniformity and Alignment. How do the key metrics used to evaluate the embedding vectors change?

RQ6. Hyperparameters. How do different hyperparameters affect the performance of the model?

4.1. Setup

**Environment.** Our experimental environment is set up on a Linux server, running Ubuntu 20.04, equipped with two Intel Xeon 4210r CPUs at 2.4Ghz (each CPU has 10 cores and 20 virtual cores), 128G memory, and 1 Nvidia RTX 3090 GPU (24G memory). The software environment is Python 3.8 and PyTorch 1.11.0 with CUDA 11.7.

**Hyperparameters.** SimCLF model is implement with two encoders. First one is TREX, a BERT based pre-trained model [35], which has been pretrained for 10 epochs. For fine-tuning, the total number of epochs is set to 40, the temperature is set to 0.07, and the dimension of the output embedding is set to 768. We use the Adam optimizer with a learning rate of 0.00001 and set the maximum input length to 512. The model is trained with a weight decay of 1e-4. And we use a batch size of 32. It is named SimCLF in this section.

The second encoder is jTrans, a BERT based pre-trained model. It embeds control flow information of binary code into Transformer-based language models. For fine-tuning, the total number of epochs is set to 2, the temperature is set
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

Figure 3: The frequency of the assembler word follows the Zipf distribution. The X-axis represents the rank of instructions in the corpus, while the Y-axis represents the logarithmic value of instruction frequencies. The graphic illustrates a long tail that has been rarely seen.

Log_frequency(instruction_frequency)

| 0  | 250 | 500 | 750 | 1000 | 1250 | 1500 | 1750 | 2000 |
|----|-----|-----|-----|------|------|------|------|------|
| 2  |     |     |     |      |      |      |      |      |
| 4  |     |     |     |      |      |      |      |      |
| 6  |     |     |     |      |      |      |      |      |
| 8  |     |     |     |      |      |      |      |      |
| 10 |     |     |     |      |      |      |      |      |
| 12 |     |     |     |      |      |      |      |      |
| 14 |     |     |     |      |      |      |      |      |

to 0.07, and the dimension of the output embedding to is set 512. The Adam optimizer is used with a learning rate of 0.000005 and the maximum input length is set to 512. We train the model with a weight decay of $1e^{-4}$ and we use a batch size of 16. It is named jTrans-SimCLF in this section.

**Metrics.** In practice, functions that are compiled from the same source code are used as similar pairs. This saves a lot of manual labelling effort, and it is also a common practice in academia [11][35]. The cosine similarity function is utilized to calculate the distance between the embedding vectors of two functions. If the distance between a pair of functions is below a certain threshold, the two functions are classified as different. The robustness of our detection model is evaluated using several mechanism. The first one is AUC-ROC. Typically, the Receiver Operating Characteristic (ROC) curve [49] of the model is plotted. The ROC curve has the false positive rate on the x-axis and the true positive rate on the y-axis. The area under the plotted ROC curve, known as the AUC (Area Under Curve) is used to assess the performance of the models. The AUC values range between 0.5 and 1, where a higher value indicates a better-performing model. MRR evaluates how well the system ranks the correct or relevant item at the top of the list. It considers the reciprocal rank of the first relevant item found for each query, and then calculates the average reciprocal rank across all queries. Recall@1 calculates the percentage of queries where the first result in the ranked list is a relevant item. It indicates how well the system is able to retrieve the most relevant item as the top recommendation.

4.2. RQ1: Statistical Laws of Assembly Code

The initial question we address revolves around the distribution of assembly code frequencies and whether pre-trained models suffer from expression collapse issues. Despite the impressive results achieved by deep neural network-based encoders in binary code similarity detection, concerns may arise regarding the performance of these models due to the complexity and lack of interpretability associated with neural networks [1]. Several articles such as [15][27] have highlighted the issue of expression collapse in pre-trained models within the field of NLP. This problem arises from the uneven distribution of word frequencies, wherein high-frequency words occur more frequently and therefore receive more gradient updates during the optimization process. Conversely, low-frequency words receive fewer updates, leading to a reduced impact on the embeddings. Consequently, the embeddings of high-frequency words tend to collapse towards a corner in the embedding space. A similar phenomenon has been observed in assembly language, as depicted in Figure 1. The embeddings of assembly tokens also seem to collapse into a corner.

So we have a conjecture that the distribution of the assembly code is also close to the Zipf distribution. To explore this hypothesis, we meticulously computed the frequency of each assembly token within the fine-tuning dataset documented in [35], encompassing a substantial corpus of approximately 300 million tokens. The ensuing analysis culminated in the generation of Figure 3, wherein the observed frequency distribution of the assembly code exhibits a notable resemblance to the characteristic pattern elicited by the Zipf distribution. We believe it is the reason that lead to the problem of expression collapse in the TREX model as shown in Figure 1.

After observing the collapse of the embedded representation at the instruction level, it is reasonable to assume that the representation at the function level is also affected. The most commonly used approach to derive fixed-size sentence embedding is to average the BERT output layer [38]. Prior to fine-tuning the TREX model, a direct adoption of average
pooling to derive function representations by utilizing the token embeddings at the last layer leads to unsatisfactory outcomes. In other words, employing average pooling directly on the individual token embeddings at the final layer does not yield favorable results. Almost all function pairs achieved a similarity score of between 0.6 and 1.0, as illustrated in Figure 4. It is notable that both similar and dissimilar function pairs exhibited cosine distances of approximately 0.6. However, after fine-tuning the model with SimCLF, we are able to successfully mitigate the anisotropy of the model [12]. The utilization of average pooling to derive function embeddings enables effective differentiation between similar and dissimilar functions, as demonstrated in Figure 5. Specifically, the cosine similarity of function pairs with identical semantics ranges from $[0.3, 1]$, whereas dissimilar functions exhibit cosine similarities between $[-1, 0.3]$.

Figure 4: As shown in the figure, the yellow bar represents similar function pairs, and blue represents dissimilar function pairs. For the original TREX model, it has been observed that the majority of pairs of functions with the same meaning tend to receive a score of approximately 0.6. Similarly, most pairs of functions with different meanings also tend to obtain a score around 0.6. Therefore, it can be concluded that the original TREX does not effectively distinguish between functions with different semantics.

Figure 5: After fine-tuning the model, the obtained embeddings can be effectively employed to discern between similar and dissimilar functions with notable accuracy.

4.3. RQ2: The Accuracy of The Model

To evaluate the accuracy of SimCLF model, we conducted a comparative analysis with various SOTA models. The results of this evaluation demonstrated that SimCLF models consistently outperformed the other models, attaining the best overall results in terms of accuracy.
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

(a) ROC curves on AMD64ARMOpenSSL Dataset. (b) ROC curves on AMD64multipleCompilers Dataset.

Figure 6: ROC curves were generated for the AMD64ARMOpenSSL Dataset and AMD64multipleCompilers Dataset, comparing the performance of different models. The curves of SimCLF are represented in blue, and they consistently outperform all the other models, as indicated by their superior performance across the evaluation metrics.

Table 1
AUC scores on AMD64ARMOpenSSL and AMD64multipleCompilers.

| Models                  | AMD64ARMOpenSSL | AMD64multipleCompilers |
|-------------------------|------------------|------------------------|
| SimCLF trained on TREX  | 0.982            | 0.987                  |
| SimCLF trained on SAFE  | 0.999            | 0.999                  |
| SAFE trained on SAFE    | 0.992            | 0.990                  |
| Gemini trained on SAFE  | 0.948            | 0.932                  |

4.3.1. Compared with SAFE model

In the comparison between SimCLF and SAFE on the dataset published by SAFE [32], we focus on two specific datasets. The first dataset is called AMD64ARMOpenSSL, which consists of 95,535 functions. All these functions have been compiled for both x86-64 and ARM architectures utilizing gcc-5.4 with four different optimization levels. The dataset used for comparison includes two open-source software repositories, namely OpenSSL-1.0.1f and OpenSSL-1.0.1u. Additionally, there is another dataset named AMD64multipleCompilers, which comprises a total of 452,598 functions. They come from the following open source software: binutils-2.30, ccv-0.7, coreutils-8.29, curl-7.61.0, gsl-2.5, libhttpd-2.0, openmpi-3.1.1, openssl-1.1.1-pre8, and valgrind-3.13.0. two experiments were conducted for each dataset. In the first experiment, our model was trained using the TREX data and then tested on the SAFE data. In the second experiment, SimCLF model was trained using both the AMD64ARMOpenSSL and AMD64multipleCompilers datasets. The results obtained from these experiments will be compared with the claims made by SAFE and Gemini [45]. The experiments were conducted with a total of two runs, and the obtained results were averaged to ensure accuracy and reliability. Figure 6 a illustrates the outcomes for the AMD64ARMOpenSSL case, revealing that SimCLF achieved a mean AUC testing score of 0.999. In comparison, SAFE obtained an AUC score of 0.992 for the same dataset. Similarly, in Figure 6 b, the results for the AMD64multipleCompilers case indicate that SimCLF attained a mean testing AUC score of 0.999, while SAFE achieved an AUC score of 0.990. Additionally, the evaluation result of Gemini [45] on the SAFE data [32] for further comparison purposes is also included. It is evident that SimCLF outperforms both the SAFE and Gemini models in terms of AUC scores on both datasets.

4.3.2. Compared with TREX Model

TREX is the state-of-the-art model proposed in [35]. It leverages a hierarchical Transformer architecture to acquire execution semantics from micro-traces in the pre-training phase. Subsequently, the authors employ InferSent [8] as a supervised embedding model to fine-tune the pre-trained TREX model. In this section, we conducted fine-tuning experiments on the TREX model using a shared dataset comprising 89,809 labelled function pairs. However, during the fine-tuning process, a subset of 22,453 positive examples from the shared dataset were selectively utilized for
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

Table 2
Results on function pairs across architectures, optimizations, and obfuscations.

| Compiler         | ARCH TREX | SimCLF | OPT TREX | SimCLF | OBF TREX | Fun2Vec | ARCH+OPT TREX | SimCLF | ARCH+OPT+OBF TREX | SimCLF |
|------------------|-----------|--------|----------|--------|----------|---------|----------------|--------|-------------------|--------|
| Binutils         | 0.998     | 0.998  | 0.993    | 0.993  | 0.998    | 0.999   | 0.975          | 0.996  | 0.974             | 0.994  |
| Coreutils        | 0.998     | 0.998  | 0.993    | 0.994  | 0.998    | 0.999   | 0.977          | 0.995  | 0.973             | 0.993  |
| Curl             | 0.995     | 0.995  | 0.989    | 0.989  | 0.997    | 0.998   | 0.972          | 0.990  | 0.972             | 0.992  |
| Diffutils        | 0.998     | 0.999  | 0.994    | 0.994  | 0.998    | 0.999   | 0.973          | 0.991  | 0.973             | 0.992  |
| Gmp              | 0.984     | 0.984  | 0.982    | 0.982  | 0.996    | 0.998   | 0.972          | 0.990  | 0.973             | 0.990  |
| Findutils        | 0.999     | 0.999  | 0.991    | 0.991  | 0.995    | 0.997   | 0.975          | 0.993  | 0.974             | 0.993  |
| ImageMagick      | 0.984     | 0.984  | 0.978    | 0.978  | 0.998    | 0.999   | 0.965          | 0.987  | 0.953             | 0.980  |
| Libmicrohttpd    | 0.996     | 0.996  | 0.987    | 0.987  | 0.993    | 0.999   | 0.965          | 0.986  | 0.959             | 0.978  |
| LibTomCrypt      | 0.983     | 0.983  | 0.978    | 0.978  | 0.998    | 0.999   | 0.963          | 0.985  | 0.949             | 0.975  |
| OpenSSL          | 0.996     | 0.996  | 0.993    | 0.993  | 0.998    | 0.999   | 0.964          | 0.994  | 0.953             | 0.978  |
| PuTTY            | 0.995     | 0.995  | 0.979    | 0.979  | 0.995    | 0.997   | 0.957          | 0.977  | 0.953             | 0.977  |
| SQLite           | 0.996     | 0.996  | 0.979    | 0.979  | 0.997    | 0.999   | 0.957          | 0.977  | 0.953             | 0.978  |
| Zlib             | 0.973     | 0.974  | 0.973    | 0.974  | 0.998    | 0.999   | 0.957          | 0.976  | 0.953             | 0.978  |
| Average          | **0.992** | **0.992** | **0.986** | **0.987** | **0.997** | **0.999** | **0.967** | **0.988** | **0.963** | **0.984** |

SimCLF model. Subsequently, the performance of SimCLF model were evaluated on the designated test dataset of the TREX model. The test dataset collects 13 popular open-source software projects. These projects include Binutils-2.34, Coreutils-8.32, Curl-7.71.1, Diffutils-3.7, Findutils-4.7.0, GMP-6.2.0, ImageMagick-7.0.10, Libmicrohttpd-0.9.71, LibTomCrypt-1.18.2, OpenSSL-1.0.1f and OpenSSL-1.0.1u, PuTTY-0.74, SQLite-3.34.0, and Zlib-1.2.11. As depicted in Table 2, function pairs were divided for each project (indicated in the first column) by employing five distinct types of partitioning strategies. (1)ARCH: the function pairs have different architectures but the same optimizations without obfuscations (2nd column). (2)OPT: the function pairs have different optimizations but the same architectures without obfuscations (3rd column). (3)OBF: the function pairs have different obfuscations with the same architectures (x64) and no optimization (4th column). (4)ARCH+OPT: the function pairs have both different architectures and optimizations without obfuscations (5th column). (5)ARCH+OPT+OBF: the function pairs can come from arbitrary architectures, optimizations, and obfuscations (6th column). A comparative analysis is conducted between SimCLF and TREX using identical test datasets, and the corresponding results are presented in Table 2. On average, SimCLF demonstrates a superior performance with an AUC score exceeding 0.984. Notably, in the specific case of partition 3, SimCLF achieves perfect performance, attaining an exceptional AUC score of 0.999. Upon analyzing the results for partitions 1, 2, and 3, it is observed that the performance of both SimCLF model and TREX model are quite similar. Due to the simplicity of the data in partitions 1, 2, and 3, both models have reached the limits of accuracy. The disparity between the two models is minimal across these partitions. However, as the complexity of the samples increases, SimCLF model demonstrates a notably superior performance compared to TREX. This observation suggests that SimCLF possesses enhanced analytical capabilities when dealing with challenging examples. Based on these findings, we can confidently conclude that SimCLF outperforms the original TREX model comprehensively in terms of overall performance.

4.3.3. Compared with jTrans Model

jTrans [42] is a novel Transformer-based approach that aims to learn representations of binary code. It is the first solution to embed control flow information of binary code into Transformer-based language models, achieved through a jump-aware representation of the analyzed binaries and a newly-designed pre-training task. In our study, we conducted fine-tuning experiments on both the jTrans model and our SimCLF model, using a shared dataset BinaryCorp which is introduced by [42]. BinaryCorp comprises two datasets, namely BinaryCorp-3M and BinaryCorp-26M. BinaryCorp-3M includes 10,265 binary programs and approximately 3.6 million functions. BinaryCorp-26M includes 48,130 binary programs and approximately 26 million functions, surpassing GNUtils’ 161,202 and Coreutils’ 76,174 functions by over 160 times and 339 times, respectively. The retrieval performance are evaluated using the two metrics MRR and Recall@1. In addition to jTrans, six other models are selected as baselines. They are Genius[13], Gemini [45], SAFE
Table 3
Results of different binary similarity detection methods on BinaryCorp-3M (Poolsize=32).

| Models         | MRR | Recall@1 | O0,03 | O1,03 | O2,03 | O0,Os | O1,Os | O2,Os | Average | O0,03 | O1,03 | O2,03 | O0,Os | O1,Os | O2,Os | Average |
|----------------|-----|----------|-------|-------|-------|-------|-------|-------|---------|-------|-------|-------|-------|-------|-------|---------|
| Gemini         | 0.388 | 0.580 | 0.750 | 0.455 | 0.546 | 0.614 | 0.556 | 0.238 | 0.457 | 0.669 | 0.302 | 0.414 | 0.450 | 0.422 |
| SAFE           | 0.826 | 0.917 | 0.958 | 0.854 | 0.927 | 0.927 | 0.902 | 0.729 | 0.869 | 0.933 | 0.766 | 0.879 | 0.880 | 0.843 |
| Asm2Vec        | 0.479 | 0.878 | 0.961 | 0.536 | 0.855 | 0.900 | 0.768 | 0.351 | 0.828 | 0.942 | 0.408 | 0.796 | 0.863 | 0.701 |
| GraphEmb       | 0.602 | 0.694 | 0.750 | 0.632 | 0.674 | 0.675 | 0.671 | 0.485 | 0.600 | 0.676 | 0.521 | 0.581 | 0.584 | 0.575 |
| OrderMatters   | 0.542 | 0.740 | 0.869 | 0.638 | 0.702 | 0.682 | 0.695 | 0.414 | 0.647 | 0.822 | 0.515 | 0.611 | 0.593 | 0.591 |
| Genius         | 0.377 | 0.587 | 0.868 | 0.437 | 0.600 | 0.627 | 0.583 | 0.243 | 0.479 | 0.830 | 0.298 | 0.490 | 0.526 | 0.478 |

Table 4
Results of different binary similarity detection methods on BinaryCorp-3M (Poolsize=10000).

| Models         | MRR | Recall@1 | O0,03 | O1,03 | O2,03 | O0,Os | O1,Os | O2,Os | Average | O0,03 | O1,03 | O2,03 | O0,Os | O1,Os | O2,Os | Average |
|----------------|-----|----------|-------|-------|-------|-------|-------|-------|---------|-------|-------|-------|-------|-------|-------|---------|
| Gemini         | 0.037 | 0.161 | 0.416 | 0.049 | 0.133 | 0.195 | 0.165 | 0.024 | 0.122 | 0.367 | 0.030 | 0.099 | 0.151 | 0.132 |
| SAFE           | 0.127 | 0.345 | 0.643 | 0.147 | 0.321 | 0.377 | 0.320 | 0.068 | 0.247 | 0.575 | 0.079 | 0.221 | 0.283 | 0.246 |
| Asm2Vec        | 0.072 | 0.449 | 0.669 | 0.083 | 0.409 | 0.510 | 0.366 | 0.046 | 0.367 | 0.589 | 0.052 | 0.332 | 0.426 | 0.302 |
| GraphEmb       | 0.087 | 0.217 | 0.486 | 0.110 | 0.195 | 0.222 | 0.219 | 0.050 | 0.154 | 0.447 | 0.063 | 0.135 | 0.166 | 0.169 |
| OrderMatters   | 0.060 | 0.319 | 0.600 | 0.075 | 0.260 | 0.233 | 0.263 | 0.040 | 0.248 | 0.535 | 0.040 | 0.178 | 0.158 | 0.200 |
| Genius         | 0.041 | 0.193 | 0.596 | 0.049 | 0.186 | 0.224 | 0.214 | 0.026 | 0.153 | 0.538 | 0.032 | 0.146 | 0.180 | 0.179 |
| jTrans-Zero    | 0.137 | 0.490 | 0.693 | 0.182 | 0.472 | 0.510 | 0.414 | 0.088 | 0.412 | 0.622 | 0.122 | 0.393 | 0.430 | 0.340 |
| jTrans         | 0.475 | 0.663 | 0.731 | 0.539 | 0.665 | 0.664 | 0.623 | 0.376 | 0.580 | 0.661 | 0.443 | 0.508 | 0.585 | 0.571 |
| jTrans-SimCLF  | 0.557 | 0.695 | 0.741 | 0.609 | 0.691 | 0.691 | 0.664 | 0.464 | 0.619 | 0.674 | 0.520 | 0.616 | 0.616 | 0.585 |

[32], Asm2Vec [11], GraphEmb [33], OrderMatters[48]. On each dataset, two function pool sizes 32 and 10,000 are used. In this test, our model is illustrated as jTrans-SimCLF. jTrans-zero is the model without finetuned.

The results of the experiments conducted on BinaryCorp-3M are presented in Table 3 and Table 4. For poolsize=32 (Table 3), jTrans-SimCLF demonstrates superior performance compared to its closest baseline competitor, with an average improvement of 0.06 for the MRR metric and over 0.09 for the recall@1 metric. When considering poolsize=10000, jTrans-SimCLF outperforms all baselines by significant margins. On average, jTrans-SimCLF surpasses its closest competitor, jTrans, by 0.041 for the MRR metric and over 0.014 for the recall@1 metric. Notably, jTrans-SimCLF achieves these results with just two training epochs, while jTrans was trained for ten epochs.

The results of our experiments conducted on BinaryCorp-26M are presented in Table 5 and Table 6. jTrans-SimCLF demonstrates superior performance compared to all baselines across all test scenarios. Specifically, for poolsize=32, jTrans-SimCLF outperforms jTrans by an average improvement of 0.02 for the MRR metric and over 0.01 for the recall@1 metric. Furthermore, when considering poolsize=10000, jTrans-SimCLF surpasses all baselines by significant margins. On average, jTrans-SimCLF outperforms its closest competitor, jTrans, by 0.018 for the MRR metric and over 0.021 for the recall@1 metric. These findings highlight the robust performance of jTrans-SimCLF in comparison to alternative approaches. Notably, jTrans-SimCLF achieves these results with just two training epochs and half of the dataset, while jTrans was trained for ten epochs on the complete dataset.

4.3.4. Vulnerability Search

In order to further assess the performance of SimCLF, SimCLF is applied to a real-world application scenario. Specifically, SimCLF is employed to conduct a search for vulnerable functions within a publicly available dataset [9]. The dataset utilized in this evaluation consists of multiple vulnerable binaries that were compiled using 11 different compilers belonging to the clang, gcc, and icc families[32]. A subset functions that consisting of functions with a length of less than 512 are selected. This subset contains a total of 2,220 functions, out of which 26 functions are identified as containing exploits. The dataset encompasses four distinct types of vulnerabilities, which are documented
### Table 5
Results of different binary similarity detection methods on BinaryCorp-26M (Poolsize=32).

| Models      | O0,O3 | O1,O3 | O2,O3 | O0,Os | O1,Os | O2,Os | Average | O0,O3 | O1,O3 | O2,O3 | O0,Os | O1,Os | O2,Os | Average |
|-------------|-------|-------|-------|-------|-------|-------|---------|-------|-------|-------|-------|-------|-------|---------|
| Gemini      | 0.402 | 0.643 | 0.835 | 0.469 | 0.564 | 0.628 | 0.590   | 0.263 | 0.528 | 0.768 | 0.322 | 0.441 | 0.518 | 0.473   |
| SAFE        | 0.856 | 0.940 | 0.970 | 0.874 | 0.935 | 0.934 | 0.918   | 0.770 | 0.902 | 0.951 | 0.795 | 0.891 | 0.891 | 0.867   |
| Asm2Vec     | 0.439 | 0.847 | 0.958 | 0.490 | 0.788 | 0.849 | 0.729   | 0.314 | 0.789 | 0.940 | 0.362 | 0.716 | 0.800 | 0.654   |
| GraphEmb    | 0.583 | 0.681 | 0.741 | 0.610 | 0.637 | 0.639 | 0.649   | 0.465 | 0.586 | 0.667 | 0.499 | 0.541 | 0.543 | 0.550   |
| OrderMatters| 0.572 | 0.820 | 0.932 | 0.630 | 0.692 | 0.771 | 0.729   | 0.417 | 0.740 | 0.903 | 0.481 | 0.692 | 0.677 | 0.652   |
| jTrans-Zero | 0.632 | 0.871 | 0.973 | 0.687 | 0.890 | 0.891 | 0.824   | 0.539 | 0.838 | 0.961 | 0.602 | 0.854 | 0.853 | 0.775   |
| jTrans      | 0.946 | 0.983 | 0.989 | 0.969 | 0.980 | 0.980 | 0.978   | 0.941 | 0.970 | 0.981 | 0.949 | 0.964 | 0.964 | 0.962   |
| jTran-SimCLF| 0.967 | 0.983 | 0.991 | 0.972 | 0.981 | 0.982 | 0.980   | 0.946 | 0.971 | 0.981 | 0.953 | 0.965 | 0.965 | 0.963   |

### Table 6
Results of different binary similarity detection methods on BinaryCorp-26M (Poolsize=10000).

| Models      | O0,O3 | O1,O3 | O2,O3 | O0,Os | O1,Os | O2,Os | Average | O0,O3 | O1,O3 | O2,O3 | O0,Os | O1,Os | O2,Os | Average |
|-------------|-------|-------|-------|-------|-------|-------|---------|-------|-------|-------|-------|-------|-------|---------|
| Gemini      | 0.072 | 0.189 | 0.474 | 0.069 | 0.147 | 0.202 | 0.192   | 0.058 | 0.148 | 0.420 | 0.051 | 0.115 | 0.162 | 0.159   |
| SAFE        | 0.198 | 0.415 | 0.696 | 0.197 | 0.377 | 0.431 | 0.386   | 0.135 | 0.314 | 0.634 | 0.127 | 0.279 | 0.343 | 0.305   |
| Asm2Vec     | 0.118 | 0.443 | 0.703 | 0.107 | 0.369 | 0.480 | 0.370   | 0.099 | 0.376 | 0.638 | 0.086 | 0.307 | 0.413 | 0.320   |
| GraphEmb    | 0.116 | 0.228 | 0.498 | 0.133 | 0.198 | 0.224 | 0.233   | 0.080 | 0.171 | 0.465 | 0.090 | 0.145 | 0.175 | 0.188   |
| OrderMatters| 0.113 | 0.292 | 0.682 | 0.118 | 0.256 | 0.295 | 0.292   | 0.094 | 0.222 | 0.622 | 0.093 | 0.195 | 0.236 | 0.244   |
| jTrans-Zero | 0.215 | 0.570 | 0.759 | 0.233 | 0.571 | 0.563 | 0.485   | 0.167 | 0.503 | 0.701 | 0.175 | 0.507 | 0.500 | 0.426   |
| jTrans      | 0.584 | 0.734 | 0.782 | 0.627 | 0.709 | 0.710 | 0.693   | 0.499 | 0.668 | 0.736 | 0.550 | 0.648 | 0.648 | 0.625   |
| jTran-SimCLF| 0.619 | 0.740 | 0.801 | 0.665 | 0.718 | 0.720 | 0.711   | 0.538 | 0.691 | 0.742 | 0.593 | 0.657 | 0.657 | 0.646   |

### Table 7
Basic information of the Vulnerability dataset

| # | Alias/Method | CVE/stats | Function name | Number |
|---|--------------|-----------|---------------|--------|
| 1 | Heartbleed   | cve-2014-0160 | tls1_process_heartbeat | 8      |
| 2 | Venom        | cve-2015-3456 | fdctrl_handle_drive_specification_command | 6      |
| 3 | WS-snmp      | cve-2011-0444 | snmp_usm_password_to_key_sha1 | 7      |
| 4 | wget         | cve-2014-4877 | ftp_syst | 5      |

in Table 7. To process the dataset, we disassembled the functions in dataset using the objdump tool and transformed the target functions into embeddings. Subsequently, these embeddings are compared with the functions known to contain exploits. In this vulnerability search experiment, SimCLF performed almost perfectly. It successfully identified all 26 vulnerable functions in the top-k results for each query. Specifically, when considering the case where the number of functions containing vulnerabilities is denoted as \( k \), SimCLF consistently discovered all \( k \) vulnerable functions within the first \( k \) results for every query. Comparatively, the SAFE model, as documented in the paper by [32], achieved a recall rate of 84% for \( k = 10 \). Meanwhile, the Gemini model achieved a lower recall rate of 55% for the same \( k \) value. This suggests that SimCLF outperformed both SAFE and Gemini in terms of recall for identifying vulnerable functions.

### 4.4. RQ3: Ablation Analysis

In this section, a quantitative analysis is employed to assess the influence of various components within the SimCLF model. With the exception of Subsection 4.4.4, all reported results in this section are derived from the following standardized settings. All models were trained utilizing the parameters outlined in Section 4.1, and subsequently evaluated on a total of 13 distinct datasets as described in Section 4.3.2. By maintaining consistent employment of these settings throughout all experiments, the precise impact of individual elements within the SimCLF framework can be accurately quantified.
4.4.1. Projection Head

Several articles have pointed out that project head can improve the performance of the model before it [7][5][17][6]. The influences of including a projection head $g(h)$ are studied in this section. $g(h)$ maps representations to the space where the contrastive loss is applied after the training process. However, after the finetuning phase, the projection head $g(h)$ is discarded and not used further. Figure 7 shows the difference between the model with and without the projection head. The embedding $z$ is used as the representation for downstream tasks.

Numerous scholarly articles [7][5][17][6] have employed the contrastive learning model as a prominent approach. While the function of the projection head, denoted as $g(h)$, has been widely acknowledged and demonstrated in various studies, such as SimCLR [5], the precise reasons for its effectiveness remain subjects of ongoing debate. However, it is noteworthy that in SimCLF, incorporating the projection head did not yield desirable outcomes. Figure 8 shows that the model’s AUC score increases slightly (on average 3%) when the model is trained with projection head $g(h)$ compared to the model without fine-tuning. Nevertheless, it is worth noting that the AUC score of the model with the projection head remains considerably lower than that of the model without the projection head. A conjecture put forward by SimCLR [5] suggests that the significance of utilizing the representation prior to the nonlinear projection lies in mitigating the loss of information induced by the contrastive model. Additionally, another plausible explanation is that [5] incorporates a vast amount of data, wherein some information may be compromised when employing the contrastive model with the projection head. However, in our experiments the amount of data is relatively small (tens of thousands). In the process of fine-tuning, the BERT-based encoder is used. Compared to ResNet-50 [5], BERT has a better encoding capability. The information lost due to contrastive learning is relatively small and could therefore be ignored.

4.4.2. Pretraining Effectiveness.

In order to delve deeper into the impact of the pre-training modules, a comparative analysis is conducted by removing the pre-training modules. Specifically, when the attention module is removed, SimCLF model reverts to a Word2Vec model. The AUC score of the model on various datasets is presented in Figure 9. The results reveal an average increase of 10% in the AUC score. This finding underscores the crucial role played by the pre-training module in enhancing model performance and highlights its significance in the overall architecture.

4.4.3. Only Finetuning The Projection Head

Although the model with the addition of the projection head could not yield good results, we also tried different network structures to find the best network. A full connection layer was added to the TREX model, and when finetuning, the pre-trained model is fixed. Only the project heads are finetuned and the projection heads are kept. Figure 10 shows that when only the projection heads are finetuned, the AUC value decreases slightly, about 0.05-0.08.
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

4.4.4. The data augmentation

To gain insight into the internal mechanisms of SimCLF, an empirical investigation was conducted to examine the correlation between data augmentation and downstream performance. All tests were performed on the jTrans-SimCLF model. The models were trained utilizing the parameters outlined in Section 4.1 and subsequently evaluated on the BinaryCorp-3M dataset.

To evaluate the efficacy of the jTrans-SimCLF model, all five types of transformations were amalgamated, and one of the combined transformations was subsequently eliminated in a stepwise manner from five to two, resulting in four experiments being conducted on the dataset. As shown in Table 8, it was observed that there was a monotonic decline in downstream performance as the number of transformations decreased. Therefore, it can be conclude that the use of data augmentation techniques is beneficial for the success of the model.

Figure 8: Comparison of AUC scores on test set between models pre-trained with and without projection head.

Figure 9: Comparison of testing AUC scores between models pre-trained with and without the pre-training modules.
4.6. RQ5: Uniformity and Alignment

The metrics, uniformity and alignment, that impact model performance is evaluated in this section. All the test are evaluated with the jTrans-SimCLF model. The models are trained using the parameters described in Section 4.1, and then tested on the BinaryCorp-3M dataset. Table 10 shows uniformity and alignment of different function embedding models along with their averaged results. jTrans-zero is the model without finetuned. jTrans is the model finetuned.
with triplet loss. In general, models which have both better alignment and uniformity achieve better performance. In general, jTrans-SimCLF demonstrated superior alignment and uniformity. Additionally, it was observed that (1) although the pre-trained jTrans-zero model displayed good alignment, its uniformity was poor; and (2) while jTrans exhibited the best uniformity, its alignment was poor. Therefore, it can be concluded that jTrans-SimCLF has the ability to simultaneously optimize both alignment and uniformity metrics.

4.7. RQ6: Temperature and Epoch

The influence of some hyperparameters such as the number of training epochs and temperature is evaluated in this section. Since both SimCLF and TREX use the same pre-trained model, the difference at the fine-tuning phase is compared. The fine-tuning data for the TREX model contains approximately 89,809 function pairs. SimCLF model requires only 22,453 functions. Although SimCLF model requires fewer samples, it converges significantly faster and better accuracy can be obtained after only about 2 rounds. As shown in the Table 11, the TREX model converges much slower. After 20 training rounds, the average AUC is still lower than that obtained by training SimCLF for 2 rounds. The results show that by avoiding the over-training problem caused by mislabelling, SimCLF converges significantly faster with less data.

The importance of the temperature $\tau$ is also evaluated. The temperature $\tau$ in the loss (Equation (4) ) is used to control the smoothness of the distribution. The smaller the $\tau$ is the sharper the distribution will be. Figure 11 shows the AUC at different temperatures. The best performance is obtained when the temperature is set to 0.07.

Figure 11: The influence of different temperatures on the loss function.

5. Discussion and Future Works

Inspired by the advances in the field of NLP, many people [35][32] are using NLP-related techniques to solve problems in the field of assembly language. Although many people use the BERT model to generate embeddings...
they all ignore the robustness of learned embeddings. In this paper, unsupervised contrastive learning is used to transform the function-level embedding learning into an instance discrimination problem. The difficulty of accurately labelling function pairs is avoid, and SimCLF achieves SOTA achievements on multiple datasets. At the same time, SimCLF shows some characteristics different from images, such as very high accuracy without project heads which merits further investigation. Other advances in contrastive learning [19] are also worth being applied in BCSD fields.

6. Conclusion

In this paper, we introduce SimCLF, a self-supervised contrastive learning framework designed to generate embeddings for binary functions, with the aim of facilitating downstream tasks. Notably, SimCLF stands out due to its simplicity, as it does not necessitate additional intricate structures and can be seamlessly integrated with any encoder. Moreover, we present the pioneering application of unsupervised contrastive learning techniques to pre-trained models within the BCSD domains.

To substantiate the efficacy of SimCLF, extensive experiments were conducted across diverse datasets, revealing compelling results. SimCLF consistently achieved state-of-the-art performance. Furthermore, our framework exhibited remarkable resilience in the few-shot setting, demonstrating its robustness even when faced with limited training data. As proponents of open research and knowledge sharing, we have made SimCLF codes publicly available, with the intention of inspiring among fellow researchers. We hope that our contributions will motivate further exploration and innovation in related domains.

References

[1] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K., 2020. Dos and don’ts of machine learning in computer security. CoRR abs/2010.09470. URL: https://arxiv.org/abs/2010.09470, arXiv:2010.09470.

[2] Baker, B., 1995. On finding duplication and near-duplication in large software systems, in: Proceedings of 2nd Working Conference on Reverse Engineering, pp. 86–95. doi:10.1109/WCRE.1995.514697.

[3] Baker, B.S., Manber, U., Muth, R., 1999. Compressing differences of executable code, in: In ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS’99).

[4] Bilenko, M., Mooney, R.J., 2003. Adaptive duplicate detection using learnable string similarity measures, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery, New York, NY, USA. p. 39–48. URL: https://doi.org/10.1145/956750.956759, doi:10.1145/956750.956759.

[5] Chen, T., Kornblith, S., Norouzi, M., Hinton, G., . A simple framework for contrastive learning of visual representations. URL: http://arxiv.org/abs/2002.05709, arXiv:2002.05709 [cs, stat]. number: arXiv:2002.05709.

[6] Chen, T., Kornblith, S., Swersky, K., Norouzi, M., Hinton, G.E., 2020a. Big self-supervised models are strong semi-supervised learners. CoRR abs/2006.10029. URL: https://arxiv.org/abs/2006.10029, arXiv:2006.10029.

[7] Chen, X., Fan, H., Girshick, R.B., He, K., 2020b. Improved baselines with momentum contrastive learning. CoRR abs/2003.04297. URL: https://arxiv.org/abs/2003.04297, arXiv:2003.04297.

[8] Conneau, A., Kiela, D., Schwenk, H., Barrault, L., Bordes, A., . Supervised learning of universal sentence representations from natural language inference data. URL: http://arxiv.org/abs/1705.02364, arXiv:1705.02364 [cs]. number: arXiv:1705.02364.

[9] David, Y., Partush, N., Yahav, E., . Statistical similarity of binaries . 15.

[10] Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 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), Association for Computational Linguistics, Minneapolis, Minnesota. pp. 4171–4186. URL: https://aclanthology.org/N19-1423.

[11] Ding, S.H.H., Fung, B.C.M., Charland, P., . 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, pp. 472–489. URL: https://ieeexplore.ieee.org/document/8835340/, doi:10.1109/SP.2019.00003.

[12] Ethayarajh, K., . How contextual are contextualized word representations? comparing the geometry of BERT, ELMo, and GPT-2 embeddings, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP), Association for Computational Linguistics. pp. 55–65. URL: https://www.aclweb.org/anthology/D19-1006, doi:10.18653/v1/D19-1006.

[13] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H., 2016. Scalable graph-based bug search for firmware images, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 480–491. URL: https://doi.org/10.1145/2977649.2977830, doi:10.1145/2977649.2977830.

[14] Gao, D., Reiter, M.K., Song, D., 2008. Binhunt: Automatically finding semantic differences in binary programs, in: Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20 - 22, 2008 Proceedings, Springer-Verlag, Berlin, Heidelberg. p. 238–255. URL: https://doi.org/10.1007/978-3-540-88625-9_16, doi:10.1007/978-3-540-88625-9_16.

[15] Gao, J., He, D., Tan, X., Qin, T., Wang, L., Liu, T.Y., a. REPRESENTATION DEGENERATION PROBLEM IN TRAINING NATURAL LANGUAGE GENERATION MOD-. 14.
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

[16] Gao, T., Yao, X., Chen, D., b. SimCSE: Simple contrastive learning of sentence embeddings. URL: http://arxiv.org/abs/2104.08821, arXiv:2104.08821 [cs]. number: arXiv:2104.08821.

[17] Grill, J., Strub, F., Altché, F., Tallec, C., Richemond, P.H., Buchatskaya, E., Doersch, C., Pires, B.A., Guo, Z.D., Azar, M.G., Piot, B., Kavukcuoglu, K., Munos, R., Valko, M., 2020. Bootstrap your own latent: A new approach to self-supervised learning. CoRR abs/2006.07733. URL: https://arxiv.org/abs/2006.07733.

[18] Haq, I.U., Caballero, J., A survey of binary code similarity. URL: http://arxiv.org/abs/1909.11424, arXiv:1909.11424 [cs]. number: arXiv:1909.11424.

[19] He, K., Fan, H., Wu, Y., Xie, S., Girshick, R., 2013. Momentum contrast for unsupervised visual representation learning , 10.

[20] Hinton, G., Vinyals, O., Dean, J., 2015. Distilling the knowledge in a neural network. arXiv:1503.02531.

[21] Hu, X., Bhatkar, S., Griffin, K., Shin, K.G., 2013. MutantX-s: Scalable malware clustering based on static features, in: Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX Association, USA, p. 178–187.

[22] Hu, X., Chiueh, T.C., Shin, K.G., 2009. Large-scale malware indexing using function-call graphs, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 611–620. URL: https://doi.org/10.1145/1653662.1653736.

[23] Huang, W., Yi, M., Zhao, X., 2021. Towards the generalization of contrastive self-supervised learning. CoRR abs/2111.00743. URL: https://arxiv.org/abs/2111.00743.

[24] Khoo, W.M., Mycroft, A., Anderson, R., Rendezvous: A search engine for binary code, in: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE. pp. 329–338. URL: http://ieeexplore.ieee.org/document/6624046/.

[25] Koo, H., Park, S., Choi, D., Kim, T., 2021. Semantic-aware binary code representation with BERT. CoRR abs/2106.05478. URL: http://arxiv.org/abs/2106.05478.

[26] Le, Q., Mikolov, T., Distributed representations of sentences and documents, 9.

[27] Li, B., Zhou, H., He, J., Wang, M., Yang, Y., Li, L., a. On the sentence embeddings from pre-trained language models, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, pp. 9119–9130. URL: https://www.aclweb.org/anthology/2020.emnlp-main.733, doi:10.18653/v1/2020.emnlp-main.733.

[28] Li, X., Yu, Q., Yin, H., PalmTree: Learning an assembly language model for instruction embedding. URL: http://arxiv.org/abs/2103.03809, doi:10.1145/34640120.3484687.

[29] Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W., 2018. Diff: Cross-version binary code similarity detection with dnn, in: Proceedings of the 33rd ACM/IEEE Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, p. 667–678. URL: https://doi.org/10.1145/3238147.3238199, doi:10.1145/3238147.3238199.

[30] Liu, C., Chen, C., Han, J., Yu, P.S., 2006. Gplag: Detection of software plagiarism by program dependence graph analysis, in: In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’06, ACM Press. pp. 872–881.

[31] Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S., 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. IEEE Transactions on Software Engineering 43, 1157–1177. doi:10.1109/TSE.2017.2655046.

[32] Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R., SAFE: Self-attentive function embeddings for binary similarity. URL: http://arxiv.org/abs/1811.05296, arXiv:1811.05296 [cs]. number: arXiv:1811.05296.

[33] Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R., 2019. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis, in: Proceedings 2019 Workshop on Binary Analysis Research, Internet Society, San Diego, CA. URL: https://www.ndss-symposium.org/wp-content/uploads/bar2019_20_Massarelli_paper.pdf, doi:10.14722/bar.2019.23020.

[34] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality, in: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, Curran Associates Inc., Red Hook, NY, USA. p. 3111–3119.

[35] Mikolov, T., Yatskevich, D., Komiske, S., Yegros-Luena, J., Sutskever, I., 2012. On the sentence embeddings from pre-trained language models, in: Proceedings of the 2012 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, pp. 9119–9130. URL: https://www.aclweb.org/anthology/2012.emnlp-main.733, doi:10.18653/v1/2012.emnlp-main.733.

[36] Pewny, J., Schuster, F., Bernhard, L., Holz, T., Rossov, C., 2014. Leveraging semantic signatures for bug search in binary programs, in: Proceedings of the 30th Annual Computer Security Applications Conference, Association for Computing Machinery, New York, NY, USA, p. 406–415. URL: https://doi.org/10.1145/2664243.2664269, doi:10.1145/2664243.2664269.

[37] Reimers, N., Gurevych, I., Sentence-BERT: Sentence embeddings using siamese BERT-networks. URL: http://arxiv.org/abs/1908.10084, arXiv:1908.10084 [cs]. number: arXiv:1908.10084.

[38] Shrivastava, A., Gupta, A., Girshick, R., 2016. Training region-based object detectors with online hard example mining, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 761–769. doi:10.1109/CVPR.2016.89.

[39] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA. p. 6000–6010.

[40] Wang, F., Liu, H., 2021. Understanding the behaviour of contrastive loss, in: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2495–2504. doi:10.1109/CVPR46437.2021.00252.

[41] Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C., 2022. jtrans: Jump-aware transformer for binary code similarity. arXiv:2205.12713.

[42] Wang, T., Isola, P., 2020. Understanding contrastive representation learning through alignment and uniformity on the hypersphere, in: Proceedings of the 37th International Conference on Machine Learning, JMLR.org.
SimCLF: A Simple Contrastive Learning Framework for Function-level Binary Embeddings

[44] Wu, Z., Xiong, Y., Yu, S.X., Lin, D., . Unsupervised feature learning via non-parametric instance discrimination, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, IEEE. pp. 3733–3742. URL: https://ieeexplore.ieee.org/document/8578491/, doi:10.1109/CVPR.2018.00393.

[45] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D., . Neural network-based graph embedding for cross-platform binary code similarity detection, 14.

[46] Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. doi:10.1109/SP.2014.44.

[47] Yan, Y., Li, R., Wang, S., Zhang, F., Wu, W., Xu, W., . ConSERT: A contrastive framework for self-supervised sentence representation transfer. URL: http://arxiv.org/abs/2105.11741, arXiv:2105.11741 [cs], number: arXiv:2105.11741.

[48] Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S., . Order matters: Semantic-aware neural networks for binary code similarity detection 34, 1145–1152. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5466, doi:10.1609/aaai.v34i01.5466.

[49] Zhang, F., Gong, T., Lee, V.E., Zhao, G., Rong, C., Qu, G., 2016. Fast algorithms to evaluate collaborative filtering recommender systems. Knowledge-Based Systems 96, 96–103. URL: https://www.sciencedirect.com/science/article/pii/S0950705115005079, doi:https://doi.org/10.1016/j.knosys.2015.12.025.

[50] Zhang, N., . Hikari – an improvement over obfuscator-llvm. URL: https://github.com/HikariObfuscator/Hikari.