applied sciences

MDPI

Article

BlockMatch: A Fine-Grained Binary Code Similarity Detection Approach Using Contrastive Learning for Basic Block Matching

Zhenhao Luo, Pengfei Wang *, Wei Xie, Xu Zhou and Baosheng Wang

College of Computer, National University of Defense Technology, Changsha 410073, China
* Correspondence: pfwang@nudt.eud.cn

Abstract: Binary code similarity detection (BCSD) plays a vital role in computer security and software engineering. Traditional BCSD methods heavily rely on specific features and necessitate rich expert knowledge, which are sensitive to code alterations. To improve the robustness against minor code alterations, recent research has shifted towards machine learning-based approaches. However, existing BCSD approaches mainly focus on function-level matching and face challenges related to large batch optimization and high quality sample selection at the basic block level. To overcome these challenges, we propose BlockMatch, a novel fine-grained BCSD approach that leverages natural language processing (NLP) techniques and contrastive learning for basic block matching. We treat instructions of basic blocks as a language and utilize a DeBERTa model to capture relative position relations and contextual semantics for encoding instruction sequences. For various operands in binary code, we propose a root operand model pre-training task to mitigate semantic missing of unseen operands. We then employ a mean pooling layer to generate basic block embeddings for detecting binary code similarity. Additionally, we propose a contrastive training framework, including a block augmentation model to generate high-quality training samples, improving the effectiveness of model training. Inspired by contrastive learning, we adopt the NT-Xent loss as our objective function, which allows larger sample sizes for model training and mitigates the convergence issues caused by limited local positive/negative samples. By conducting extensive experiments, we evaluate BlockMatch and compare it against state-of-the-art approaches such as PalmTree and SAFE. The results demonstrate that BlockMatch achieves a recall@1 of 0.912 at the basic block level under the cross-compiler scenario (pool size = 10), which outperforms PalmTree (0.810) and SAFE (0.798). Furthermore, our ablation study shows that the proposed contrastive training framework and root operand model pre-training task help our model achieve superior performance.

Keywords: binary code similarity detection; basic block matching; contrastive learning

Citation: Luo, Z.; Wang, P.; Xie, W.; Zhou, X.; Wang, B. BlockMatch: A Fine-Grained Binary Code Similarity Detection Approach Using Contrastive Learning for Basic Block Matching. *Appl. Sci.* **2023**, *13*, 12751. https://doi.org/10.3390/app132312751

Academic Editor: Bang Wang

Received: 24 August 2023
Revised: 13 October 2023
Accepted: 23 November 2023
Published: 28 November 2023

1. Introduction

Binary code similarity detection (BCSD) is a pivotal technique in the spheres of computer security and software engineering. Given an unknown binary file, the BCSD tools extract features of binary code and search corresponding similar code in the repositories. Its broad applications span various field of software analysis, including software plagiarism [1], malware analysis [2–6], and bug search [7–18].

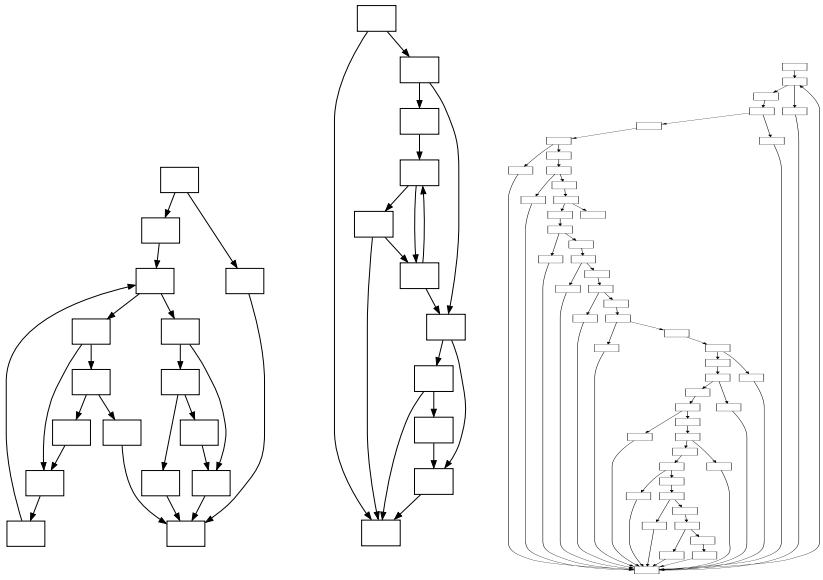
Given the significance of BCSD techniques, numerous approaches have been proposed to work on a variety of BCSD scenarios. Traditional BCSD approaches [8,18–22], which predate the incorporation of machine learning (ML) into BCSD tasks, depend heavily on specific features such as control flow graphs (CFGs), raw bytes, instructions, basic block counts, and strings for detecting similarities in binary code. For example, CFG-based approaches [20] transfer binary code into CFGs and calculate their similarities based on their graph structure, nodes, and edges. These approaches necessitate rich experience and expert knowledge to determine feature weights and are also sensitive to code alterations.

Even minor changes in binary code can cause substantial changes in these features, thereby challenging traditional BCSD methods.

In recent years, researchers have gravitated towards machine learning-based solutions to perform the BCSD tasks. Current state-of-the-art BCSD approaches [10,13,15,23–25] leverage machine learning (ML) techniques and utilize trained neural networks to comprehend the semantics of functions based on their instructions. These semantics are embedded onto normalized high-dimensional embeddings within a hyper-sphere, and we measure the similarities of binary code according to the proximity of their embeddings within the hyper-sphere. Approaches such as Genius [26] and Gemini [27] manually select the statistical features of basic blocks and employ graph neural networks (GNNs) to generate function embeddings for function matching, while jTrans [13], VulHawk [10], and SAFE [25] treat instructions of function as a language and employ Natural Language Processing (NLP) techniques to automatically extract semantics and generate function embeddings to compute function similarity.

Presently, BCSD approaches mainly focus on function-level BCSD tasks. However, due to the fact that binary functions are typically composed of many basic blocks, identifying corresponding basic blocks or code snippets in two similar functions still requires a significant amount of work, making it a challenging problem.

In binary files, basic blocks refer to sequences of instructions that have a single entry point and a single exit point. A function with complex functionality usually consists of a large number of basic blocks. Figure 1 illustrates three functions sourced from the same source code (function `register_state` in the `gawk` project) but different compilation options. Despite these functions being homologous, matching similar basic blocks or code snippets among them remains a time-consuming and challenging problem, considering their complex CFGs and numerous basic blocks. Thus, a fine-grained BCSD approach at the basic block level becomes urgent to solve the last-mile problem of BCSD tasks regarding matching among numerous basic blocks. When conducting BCSD at the basic block level, the following challenges need to be addressed.



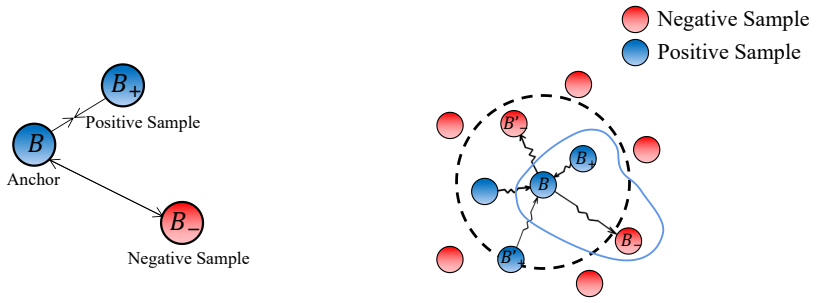
(**a**) (**b**) (**c**)

Figure 1. Functions compiled with different compilation options. They are all from the same source code of register_state in the *gawk* project, (**a**) register_state O0, (**b**) register_state O1, (**c**) register_state obfuscated.

The existing BCSD approaches employ Triplet Loss [28] to optimize the parameters of models based on partial samples [13,29]. However, this way has potential drawbacks as it may lead to suboptimal optimization and overlook better models. Triplet loss is a

widely-used training objective function, aiming to minimize the distance between embeddings of similar binary code while maximizing the distance for dissimilar ones. As shown in Figure 2a, an anchor represents an input basic block B, a positive sample B_+ denotes a similar block to the anchor, and a negative sample B_- represents a dissimilar one. During the training, the triplet loss objective brings the embeddings of block B and block B_+ closer in the hyper-sphere, while ensuring a distinct separation between B and B_-. Nevertheless, in real-world BCSD tasks, given an anchor basic block, there are numerous similar basic blocks and dissimilar basic blocks. As shown in Figure 2b, the triplet loss function focuses primarily on the local positive/negative samples and lacks a more comprehensive optimization, resulting in slow convergence and even difficulty in converging to the desired parameters.

Furthermore, selecting representative positive/negative samples from a large number of basic blocks is challenging, and the quality of the positive/negative samples seriously affects the effectiveness of model training. In BCSD tasks, easy samples represent basic blocks with distinguishable features that facilitate accurate predictions or classifications, while hard samples present challenges and complexities for the model predictions. In Figure 2b, with B as the anchor sample, B_+ and B_- are an easy positive sample and an easy negative one, while B'_+ and B'_- are a hard positive sample and a hard negative one, respectively. As depicted in Figure 2b, with B as the anchor sample, B_+ and B_- represent easy positive and easy negative samples, respectively, while B'_+ and B'_- represent hard positive and hard negative samples. Easy samples are typically associated with low prediction errors and contribute less to the overall loss calculations, while hard samples tend to result in higher prediction errors and have a greater impact on the overall loss calculations.



(**a**) (**b**)

Figure 2. An example of BCSD model training. (**a**) An example of triplet loss, and (**b**) the actual situation.

In this paper, we introduce a novel approach called `BlockMatch`for fine-grained Binary Code Similarity Detection (BCSD) that utilizes Natural Language Processing (NLP) and contrastive learning techniques to match similar basic blocks between complex functions for the last-mile problem of BCSD tasks. Basic blocks in binary files consist of various instructions. We first treat binary code as a language and use a DeBERTa model to capture position relations and contextual semantics for instruction sequence encoding. We then use a mean pooling layer to generate basic block embeddings for binary code similarity detection. To solve the aforementioned challenges, we propose a training framework based on contrastive learning [30,31]. We adopt NT-Xent loss [31] as our objective functions for model training to address the aforementioned challenges. The NT-Xent loss function supports larger sample sizes for model training, which mitigates slow convergence optimization-less problems caused by local limited positive/negative samples. We also propose a block augmentation model to generate high-quality samples for training effectiveness improvements.

In summary, we have made the following contributions:

- We propose a novel fine-grained BCSD approach based on NLP technique for basic block matching, namely `BlockMatch`. It first uses an NLP model to capture position relations and contextual semantics for instruction sequence encoding. Then, we use

a mean pooling layer to generate basic block embeddings for binary code similarity detection. `BlockMatch` can be used to solve the last-mile problem of BCSD tasks regarding matching among basic blocks.
- We propose a training framework based on contrastive learning resolving the small sample size training problem in BCSD tasks, which uses a block augmentation model to generate multiple contrastive positive/negative samples for model training and uses the NT-Xent loss objective function for parameter optimization.
- We implement `BlockMatch` and evaluate it with extensive experiments. The experiments show that `BlockMatch` outperforms the state-of-the-art approaches PalmTree and SAFE. The Ablation study shows the NT-Xent loss benefit and that our model achieves better performance than the triplet loss.

2. Related Work

This section offers a concise overview of additional related work concerning binary code similarity detection.

Generally, BCSD is geared towards quantifying the similarity of two binary code snippets extracted from executable files. As outlined in the BCSD literature [15,32,33], there are four types of binary code similarity, (1) literal identity, (2) syntactic equivalence, (3) functional equivalence, and (4) identical or logically similar source code. The current approaches [10,15,17,23,25,29,34–37] are focused on the fourth type to identify whether given binary code snippets are from the same or logic similar source code.

Significant research has been conducted in the field of binary code similarity detection. Traditional approaches, such as those proposed by Luo et al. [1], Ming et al. [2], Myles et al. [38], Jang et al. [39], and Pewny et al. [22], utilize statistical, syntax, and structural features to identify similar binary code fragments. However, these approaches require extensive expertise and knowledge to effectively select features and assign appropriate weights for BCSD tasks. DiscovRE [40], Genius [26], and GitZ [19] employ features that are robust to small changes derived from statistical, syntactic, and structural analysis to calculate binary code similarity. Trex [24] introduces a transformer-based model that utilizes micro-traces comprising instructions and dynamic values to capture function execution semantics for BCSD tasks. Drawing inspiration from NLP techniques, many researchers [10,13,15,23,37,41,42] introduce language models to automatically extract semantic features of binary code for BCSD tasks. For example, Ding et al. [15] propose Asm2Vec using the Distributed Memory Model of Paragraph Vectors (PV-DM) model [43] to represent binary functions as high-dimensional numerical embeddings to detect similar binary code. PalmTree [23] applies a transformer-based NLP model, BERT [44], for assembly instructions to measure similarity of binary code. Wang et al. [13] propose a transformer-based approach, named jTrans, to capture CFG structures using jump-aware representation for binary code similarity detection. These approaches are mainly working for functions of binary files. However, matching corresponding code snippets/basic blocks in two similar functions still requires a significant amount of effort, considering the complexity of their CFGs and the abundance of basic blocks involved. Thus, a fine-grained BCSD approach at basic block level becomes urgent to solve the last-mile problem of BCSD tasks regarding matching among numerous basic blocks.

Basic block similarity matching, as a fine-grained BCSD scenario, is more specific in identifying which basic blocks are similar compared to a function-level BCSD scenarios. At the same time, it also faces a larger number of matching samples. Current approaches represent binary code as high-dimensional embeddings on a hyper-sphere, facilitating the search for similar binary code based on cosine distance. Consequently, the primary training objective is to minimize the distance between embeddings of similar binary code while maximizing the distance between embeddings of dissimilar binary code. The existing approaches [13,29] utilize Triplet Loss [28] to optimize the parameters of models based on triplet samples (i.e., anchor sample, positive sample, and negative sample). The quality of these samples profoundly affect the effectiveness of model training. However, BCSD at

the basic blocks is a multi-class problem with thousands (or even more) of different basic blocks, and the different basic blocks are not completely different from each other, which makes it difficult to select representative samples to compute triplet loss for optimization. SimCLR [31] proposes a NT-Xent loss for the large-batch contrastive learning of visual representations and shows high performance for multi-class prediction in image classification. It support multiple negative samples and positive samples to calculate training loss, avoiding bias from local samples.

Compared with the above approaches, our proposed `BlockMatch` is a fine-grained approach for BCSD tasks at the basic block level, which uses an NLP model based on DeBERTa [45] to automatically capture position relations and contextual semantics for basic block embeddings. Considering the thousands of different basic blocks, we propose a contrastive training framework for BCSD tasks. It first uses a basic block augmentation model to generate multiple high-quality samples, and then adopts NT-Xent loss [31] as our objective functions to train our model with large-sized samples, allowing it to converge faster and achieve superior performance in BCSD tasks.

3. Design

To mitigate the problems discussed in Section 1, we propose a novel Binary Code Similarity Detection (BCSD) approach called `BlockMatch` to solve the last-mile problem of BCSD tasks regarding matching among basic blocks. Figure 3 illustrates the overall architecture of `BlockMatch`, encompassing two key modules that enable its functionality:

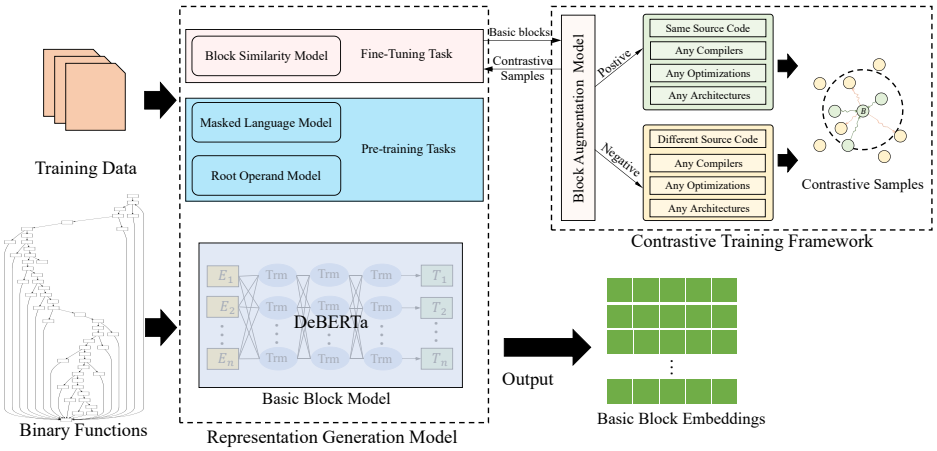


Figure 3. The overview of BlockMatch, which consists of a representation generation model and a contrastive training framework.

Representation Generation Model. This module is responsible for generating embeddings for each basic block present in the input functions. Initially, we extract basic blocks from the input functions and employ disassembly tools to convert machine code into instruction sequences for each basic block. Subsequently, we utilize a basic block model based on DeBEATa [45] to capture the semantics of these basic blocks, thereby producing corresponding embeddings. To train our model, we employ a masked language model and a novel root operand model for pre-training. We then use a block similarity model to fine-tune our model, ensuring that similar basic block embeddings are closer together.

Contrastive Training Framework. This module aims to generate diverse and high-quality samples for training the model, employing the NT-Xent loss objective function [31] to optimize the model parameters. During the training phase, given training basic blocks, we propose a block augmentation model to generate positive samples and negative samples based on their metadata. Specifically, for positive samples, the block augmentation model generates basic blocks originating from the same source code lines, but compiled using different compilers, architectures, and optimizations compared to the anchor basic block. Conversely, for negative samples, the block augmentation model generates basic blocks

with different I/O emulation results from different source code lines in relation to the anchor function. Subsequently, we utilize the NT-Xent loss objective function to optimize the representation generation model by considering their block similarities between large-batch samples.

3.1. Representation Generation Model

The representation generation model aims to generate basic block embeddings that capture relative positions, contextual information, and instruction semantics. To accomplish this, we consider basic blocks as language sentences, where their instructions are treated as words within these sentences. We leverage a customized transformer-based language model with disentangled attention [45] to capture these features and construct comprehensive representations for BCSD tasks.

The first step in the process is raw feature extraction. We extract the basic blocks from the provided binaries and employ IDA Pro [46], a disassembly tool, to convert machine code into microcode instruction sequences for these basic blocks. It is worth noting that alternative disassembly tools and intermediate representations also work.

3.1.1. Basic Block Model

To automatically generate semantic features for each basic block, we employ a basic block model based on DeBEATa [45] to capture the semantics of the instruction sequences in basic blocks. DeBERTa [45] is a member of the BERT family [44] and distinguishes itself from the conventional BERT model by employing disentangled attentions within each transformer encoder layer. Compared with BERT [44] and RoBERTa [47], DeBERTa utilizes disentangled attentions to effectively capture both the relative relations between positions and the content being analyzed, which helps our model better understand the connections between instructions in the basic blocks. Figure 4 illustrates the architecture of our basic block model, which is composed of stacked DeBERTa encoders to build embeddings. For each input sequence, we consider it as a language sentence and tokenize it into token objects. These tokens are then mapped to corresponding token embeddings. Subsequently, we feed these token embeddings into the stack of the DeBERTa encoder to generate hidden states, with the last hidden states representing the instruction semantics. Finally, we use a mean pooling on the last hidden states to generate the embedding of the basic blocks. During training, we feed the last hidden states into various training task heads to optimize our model.

3.1.2. Pre-Training Tasks

To facilitate the large-scale training of our model, we employ two self-supervised pre-training tasks: the Masked Language Model (MLM) [44] and the Root Operand Model (ROM).

Masked Language Model. The MLM task involves completing fill-in-the-blank tasks, enabling the model to learn how to predict masked tokens based on the surrounding context tokens. By performing the MLM task, the model gains a deeper understanding of the relations between instructions and positions. In MLM training, given a token sequence $\boldsymbol{X} = \{x_i | i \in (0, n)\}$, 15% of the tokens in each input sequence are randomly masked. Among the masked tokens, 80% of them are replaced with a special `[MASK]` token, 10% of them are replaced with other tokens, and the remaining 10% remain unchanged. This is carried out to create a masked sequence denoted as $\tilde{\boldsymbol{X}}$. We then input $\tilde{\boldsymbol{X}}$ into the basic block model and feed the output into an MLM head to reconstruct $\boldsymbol{X}$ by predicting the masked tokens $\tilde{x}$ conditioned on $\tilde{\boldsymbol{X}}$. The loss function for the MLM task is defined as follows:

$$\mathcal{L}_{MLM}(\theta_1, \theta_2) = -\log p_{\{\theta_1, \theta_2\}}(\boldsymbol{X} | \tilde{\boldsymbol{X}}) \tag{1}$$

Here , θ_1 and θ_2 correspond to the parameters of the basic block model and the MLM head, respectively. The pre-training task is considered complete when the training loss decreases to a stable interval, indicating convergence.

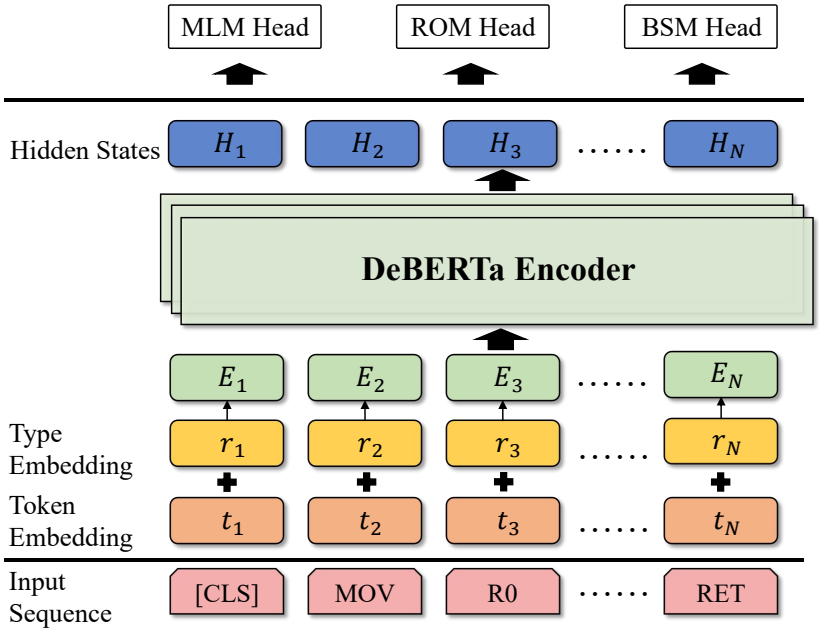


Figure 4. The architecture of the basic block model.

Root Operand Model. The ROM task aims at summarizing the root semantics of each operand, allowing the model to mitigate missing semantics for Out-of-Vocabulary words. OOV words are those that are absent from the language model's vocabulary. These are the words that are unknown to the model and bring challenges to semantic extraction. Through the ROM task, we enable the model to comprehend the intrinsic semantics of operands. When dealing with OOV operands, the model replace OOV operands with their root operands rather than completely losing their semantics. In ROM training, given a token sequence $\boldsymbol{X} = \{x_i | i \in (0, n)\}$, we generate its root operand sequence denoted as $\boldsymbol{X_r}$. We then input $\boldsymbol{X}$ into the basic block model and feed the output into an ROM head to predict their root operand x_r. The loss function for the ROM task is defined as follows:

$$\mathcal{L}_{ROM}(\theta_1, \theta_3) = -\log p_{\{\theta_1, \theta_3\}}(\boldsymbol{X_r} | \boldsymbol{X}) \tag{2}$$

Here, θ_1 and θ_3 correspond to the parameters of the basic block model and the ROM head, respectively. The pre-training task is considered complete when the training loss decreases to a stable interval, indicating convergence.

The pre-training loss function of the block semantic model is the combination of loss functions:

$$\mathcal{L} = \mathcal{L}_{MLM} + \mathcal{L}_{ROM} \tag{3}$$

3.2. Contrastive Training Framework

To fine-tune our basic block model, we propose a contrastive training framework, which is to generate diverse and high-quality samples for large-batch training. We first utilize the block augmentation model to generate diverse and high-quality positive samples and negative samples, and then construct large contrastive batches for the fine-tuning task Block Similarity Model (BSM) using the NT-Xent loss objective function to optimize the model parameters, ensuring that similar basic block embeddings are closer together.

3.2.1. Block Augmentation Model

To automatically generate diverse samples for given basic blocks, we use a block augmentation model to randomly construct similar basic blocks and dissimilar basic blocks. We first label our similar basic blocks and dissimilar basic blocks, and then perform block augmentation on the given anchor samples to generate their training samples.

Similar Basic Block Labeling. In binary files, basic blocks do not have names (like function names) to indicate whether or not two basic blocks are similar. Manual labeling basic block similarities requires a significant amount of time and effort, so we propose a method based on Debugging with Attributed Record Formats (DWARF) information for automatically labeling basic block similarities. DWARF is a debugging file format used by many compilers and debuggers to support source-level debugging [48], designed to describe compiled procedural languages such as `C` and `C++`. The DWARF provides information to indicate the source files and lines of basic blocks, which helps us to identify the basic block similarity. As shown in Figure 5, the basic block `Block1` is from the binary `vmlinux` compiled with optimization option `O2` on the `x86-64` architecture. Its DWARF information indicates `Block1` compiled from lines 172–173 of the source code file `lsm_audit.c`. The basic block `Block2` shares the same source code lines with `Block1`, so they are similar basic blocks with similar semantics.

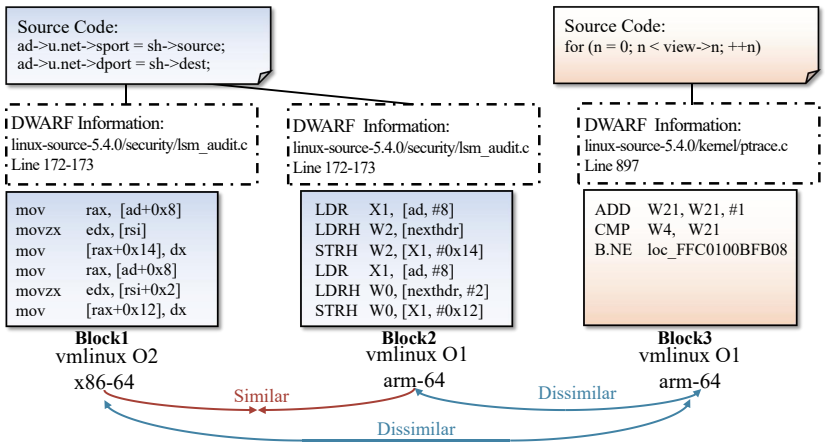


Figure 5. Basic block dataset generation based on DWARF information.

Dissimilar Basic Block Labeling. One problem arises when dealing with dissimilar basic blocks: two basic blocks from different source code have a probability of being semantically similar. In other words, different source code lines are a necessary but not sufficient condition for dissimilar basic blocks. Incorrectly labeled data will poison the dataset and impact the accuracy of the trained model. To address this problem, we use strict rules to identify dissimilar basic block pairs. Given two basic blocks, we label them as dissimilar pairs if they come from different source codes and their I/O emulation results are different. We use `Unicorn`, a multi-architecture CPU emulator framework, to emulate instructions of basic blocks. Given two basic blocks from the same architectures (e.g., `Block2` and `Block3` in Figure 5), we randomly generate their initial emulation environments (i.e., registers and stacks). For the same initial environments, the emulations of two blocks get different final environments, which indicates they have different operations on data, reflecting semantic differences in the binary code. Thus, we label `Block2` and `Block3` as dissimilar basic block pairs. Since `Block2` and `Block3` are dissimilar, considering the similarity between `Block1` and `Block2`, it can be inferred that `Block1` and `Block3` are also dissimilar. One emulation result may be accidental equivalence, so we emulate blocks multiple times with different emulation environments.

Block Augmentation. We perform block augmentation on given anchor samples in the following four aspects: source code, compiler, optimization options, and architecture. Algorithm 1 shows the details of our block augmentation model.

Algorithm 1: Block Augmentation Algorithm

Input: A basic block B, the number of positive samples N_{pos}, and the number of negative samples N_{neg}.

Output: The positive sample set S_{pos} and the negative sample set S_{neg}.

```
1  Function Augmentation (block B, flag isPositive, num N)
2      S_ret ← { } ;
3      while True do
4          if isPositive then
5              BlockSource ← GetBlockSource(f);
6          else
7              BlockSource ← DiffBlockSource();
8          compiler ← SelectCompiler();
9          opt ← SelectOpt();
10         arch ← SelectArch();
11         B' ← ConstructBlock(BlockSource, arch, opt, compiler) ;
12         if not isPositive and Emulation(B) == Emulation(B') then
13             continue ;
14         if isPositive and GetMetaData(B) == GetMetaData(B') then
15             continue ;
16         S_ret ← S_ret ∪ {B'} ;
17         if len(S_ret) >= N then
18             break ;
19     return S_ret ;
20 isPositive ← True ;
21 S_pos ← Augmentation(B, isPositive, N_pos) ;
22 isPositive ← False ;
23 S_neg ← Augmentation(B, isPositive, N_neg) ;
24 Output S_pos, S_neg;
```

To generate positive and negative samples for a given basic block B, we employ the *Augmentation* function. The parameters of *Augmentation* determine the quantity of positive and negative samples generated. When generating positive samples, our objective is to produce basic blocks that share the same semantics as basic block B, while exhibiting distinct instruction sequences. To this end, we maintain the source code lines without alteration (as shown in Line 5 of Algorithm 1), and introduce variations in the compiler, optimization techniques, and architecture employed, ensuring that the augmented basic block differs from the original basic block B. Consequently, the augmented basic blocks possess identical semantics to basic block B, but display different instructions. These augmented basic blocks pose challenges for the representation generation model, contributing significantly to the loss calculations during model training and resulting in accelerated convergence. When generating negative samples, we randomly pick basic blocks with distinct source code lines and I/O emulation results in comparison to the anchor basic block B. Subsequently, $N_{pos} + N_{neg} + 1$ contrastive samples are constructed for model training.

3.2.2. Training Objective

The objective of the fine-tuning task Block Similarity Model (BSM) is to make similar basic block embeddings closer together. Here, we use the NT-Xent loss objective function to optimize the model parameters. Given a large batch of samples, we embed them into a hyper-sphere and use cosine distance to measure their similarity scores.

Figure 6 provides an illustrative example of large batch contrastive training. During the training phase, the objective is to maximize the cosine similarity scores between embeddings

from similar function samples and simultaneously minimize the cosine similarity scores between embeddings from dissimilar function pairs. The batch is arranged in the order of the anchor sample, positive samples, and negative samples. We utilize the normalized temperature-scaled cross-entropy loss (NT-Xent) [31] as the training objective to compute the loss.

$$N = 1 + N_{pos} + N_{neg}$$
$$\mathcal{L} = -\sum_{i=1}^{N_{pos}} \sum_{j=1}^{N_{pos}} \log \frac{\exp(\mathbb{1}_{[k \neq i]} \cos(e_i, e_j)/\tau)}{\sum_{k=N-N_{neg}}^{N} \exp(\cos(e_i, e_k)/\tau)} \quad (4)$$

Here, N_{pos} and N_{neg} are the numbers of positive and negative samples, respectively. The function $\cos(\cdot)$ denotes the cosine similarity score calculation, while $\mathbb{1}$ is an indicator function that indicates 1 when its condition is true and 0 otherwise. In the NT-Xent loss function, the number of samples N consists of one anchor sample, N_{pos} positive samples, and N_{neg} negative samples. In loss $\mathcal{L}$, the numerator of the fraction computes the exponential of the similarities between different positive sample embeddings e_i and e_j. The denominator sums up the exponentials of the cosine similarities between positive sample embeddings e_i and negative sample embeddings e_k, where k ranges from $N - N_{neg}$ to N. The hyper-parameter τ plays a crucial role in controlling the temperature to help the model learn from hard negatives. We compute the gradient based on the loss $\mathcal{L}$ and use the Adam optimizer to optimize the model parameters.

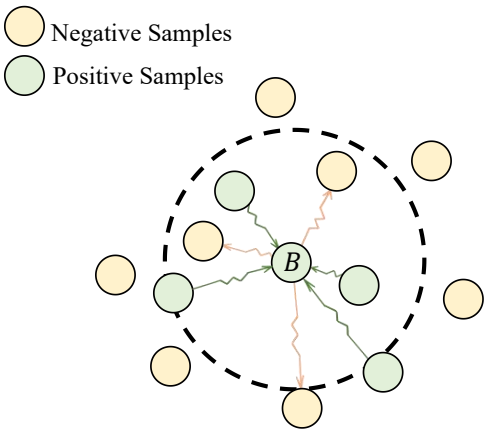


Figure 6. An illustration of contrastive training.

4. Evaluation

In this section, we evaluate `BlockMatch` and answer the following research questions:

- **RQ.1** Given two basic blocks, can `BlockMatch` identify similar basic block pairs?
- **RQ.2** Given a basic block, can `BlockMatch` retrieve similar basic blocks from a group of functions?
- **RQ.3** How much does each component contribute to the performance of `BlockMatch`?

4.1. Implementation and Setup

We developed the `BlockMatch` using PyTorch 1.13.0 [49], where the implementation of DeBERTa is based on Transformers [50]. We carefully determine the values of key hyper-parameters to strike a balance between efficiency and performance. In the representation generation model, we use a six-layer DeBERTa model with eight attention heads, where generated embeddings are 256 dimension numeric vectors. In the contrastive training framework, we set the number of positive samples, denoted as N_{pos}, to 10. Similarly, we assign a value of 20 to the number of negative samples, represented by N_{neg}. Additionally, we set the hyper-parameter τ to 0.1 in the NT-Xent loss function. Our experiments are conducted on a desktop computer running Windows 10, equipped with an Intel Core i9-10920X CPU, 64GB RAM, and an NVIDIA RTX 3090 GPU.

4.1.1. Baselines

For the purpose of comparison, we select the following state-of-the-art approaches as baselines. We download their original implementations for evaluation:

- PalmTree [23]: This method represents one of the state-of-the-art BCSD approaches. It leverages pre-trained models based on BERT [44] to generate semantic embeddings for binary code. We download its source code from the available link, https://github.com/palmtreemodel/PalmTree (accessed on 8 June 2022).
- SAFE [25]: This method employs a word2vec model [51] along with a recurrent neural network to generate function embeddings for BCSD tasks. We download its source code from GitHub, https://github.com/facebookresearch/SAFEtorch (accessed on 16 August 2023).

4.1.2. Benchmarks

Our dataset is used to evaluate the performance of `BlockMatch` and baselines at the basic block level. We construct the dataset using six projects, i.e., `GoAhead-5.2.0` [52], `CoreUtils-8.30` [53], `OpenSSL-1.1.1l` [54], `DiffUtils-3.5` [55], `Linux-5.4.0` [56], and `SQLite-3` [57], where `SQLite-3`, `Linux`, `GoAhead`, and `OpenSSL` are widely used in various software, and `CoreUtils` and `DiffUtils` are popularly used as benchmarks in the current BCSD research [10,23,41]. We compile them using two compilers (`GCC` and `Clang`) with six optimization levels (`O0`, `O1`, `O2`, `O3`, and `Os`, `Ofast`) on the `x86-64` architecture. The dataset includes a total of 3,367,762 basic blocks.

To evaluate the performance of `BlockMatch` on previously unseen projects, we adopt a leave-one-out strategy. Specifically, we randomly select one project to be excluded from the training phase. The functions from the rest of the projects are then divided into three distinct datasets of basic blocks for training, validation, and testing based on 10-fold cross-validation. During the pre-training phase, we conduct unsupervised pre-training on the MLM and ROM tasks using all of basic blocks from the training dataset. Pre-training is considered complete when the loss stabilizes and converges. During the fine-tuning phase, we use the Block Argumentation Model to generate contrastive samples for the basic blocks in the training set and perform self-supervised fine-tuning using BSM tasks. During the evaluation process, we utilize the testing dataset to conduct two different tasks including: cross optimization levels (`XO`) and cross compilers (`XC`). By incorporating these subsets into our evaluation phase, we can comprehensively evaluate the performance of `BlockMatch` across various scenarios.

4.1.3. Metrics

We use the following metrics to measure the performance of `BlockMatch` and baselines:

- **ROC**, short for Receiver Operating Characteristic, is a commonly used evaluation measure in various fields. It provides a comprehensive analysis of the trade-off between the true positive rate (TPR) and the false positive rate (FPR) at various decision thresholds. An ROC curve is a convenient visual metric for evaluating classification models. Better performance is indicated when the ROC curve is closer to the left-hand and top border.
- **AUC** stands for Area Under the ROC Curve. It represents the probability that a randomly chosen positive sample will have a higher predicted probability than a randomly chosen negative sample. A high AUC suggests the evaluated model achieves a high TPR with a low FPR. A perfect classifier will have an AUC value of 1, indicating a clear separation between the similar samples and dissimilar samples.
- **Recall@K** is a ratio of the number of correctly retrieved similar basic blocks in the top-K candidate basic blocks to the total number of similar basic blocks in the dataset. A higher recall@K value indicates that the model has successfully retrieved a larger proportion of similar basic blocks within the top K candidates, indicating better performance.

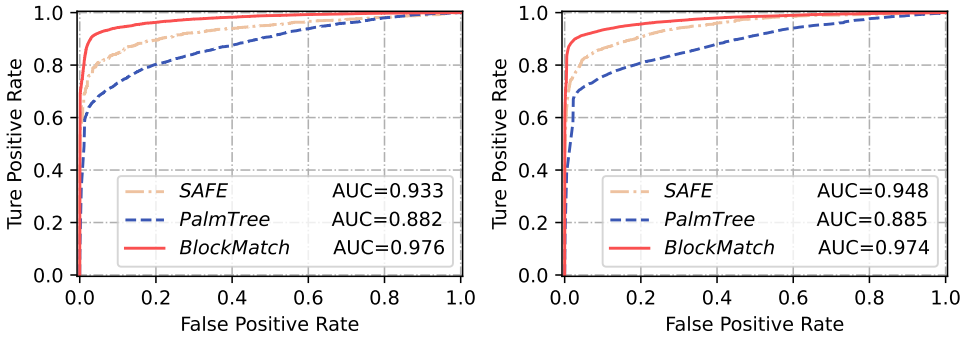
- **MRR** stands for Mean Reciprocal Rank, which measures the average reciprocal rank of the first similar basic block for the given candidates. It is calculated as the inverse of the rank of the first similar basic block, with a value of 1 indicating that the first result is correct.

In our experiments, given a basic block, we use models to find basic blocks that are similar to the given one from a function composed of a large number of basic blocks. Here, we place greater emphasis on the relevance and recalls of the top-k candidates to the given basic block, so we use recall@K and precision@K instead of recall or precision.

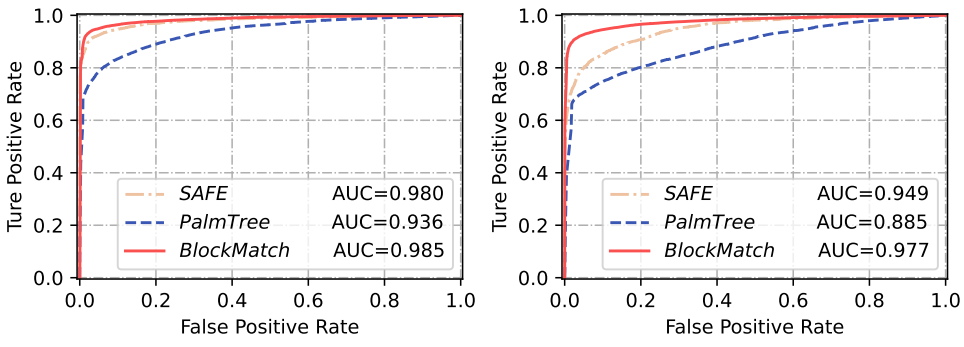
4.2. One-to-One Basic Block Matching

We conduct performance evaluations for `BlockMatch` and other baseline methods targeting one-to-one basic block matching. Given two basic blocks, the evaluated models give their similarity scores. Note that source code lines are only utilized for the ground truth construction, not for prediction.

The ROC curves of `BlockMatch` and other comparative approaches across different optimizations and compilers are displayed in Figures 7 and 8. The following findings can be derived from the results: (1) `BlockMatch` outperforms SAFE and PalmTree across different compilers and optimization options. For example, as illustrated in Figure 7f, `BlockMatch` achieves an AUC of 0.984, superseding SAFE and PalmTree, which achieve 0.931 and 0.889, respectively. (2) In the cross-compiler scenario, `BlockMatch` demonstrates strong robustness across different compilers (`Clang` and `GCC`). On the contrary, PalmTree and SAFE are prone to underperform due to compiler differences. (3) For unseen projects in the training phase, `BlockMatch` maintains a promising AUC of 0.920 (`DiffUtils`) in the cross-compiler scenario, which is comparable to the average AUC (0.942) of trained projects. (4) Compared with SAFE and PalmTree, `BlockMatch` achieves more stable performance. With the help of a contrastive training framework, `BlockMatch` has a better capability to distinguish basic block similarity.

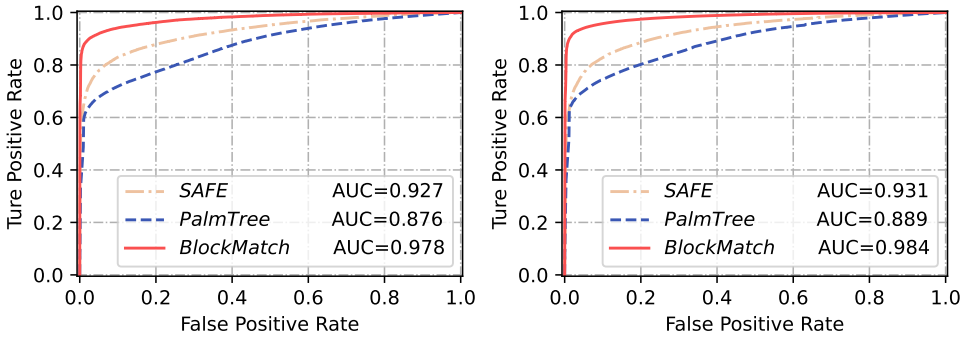


(**a**) goahead (**b**) diffutils



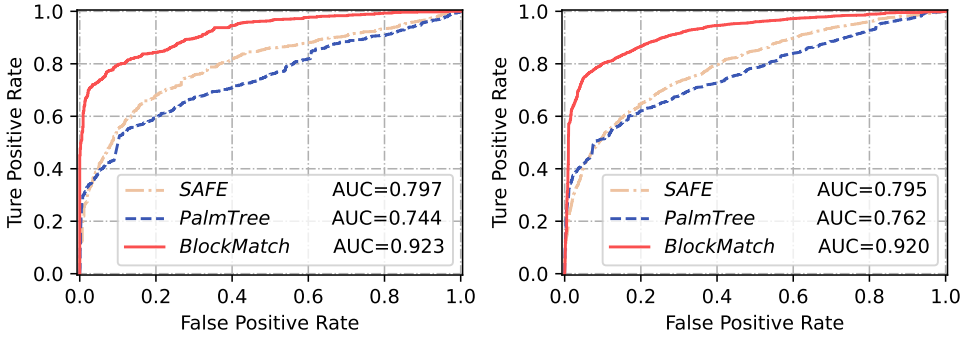
(**c**) openssl (**d**) coreutils

Figure 7. *Cont.*

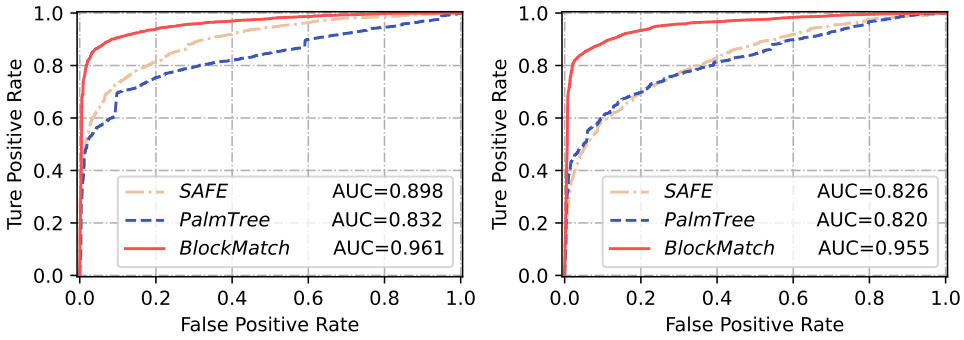


(**e**) sqlite (**f**) linux

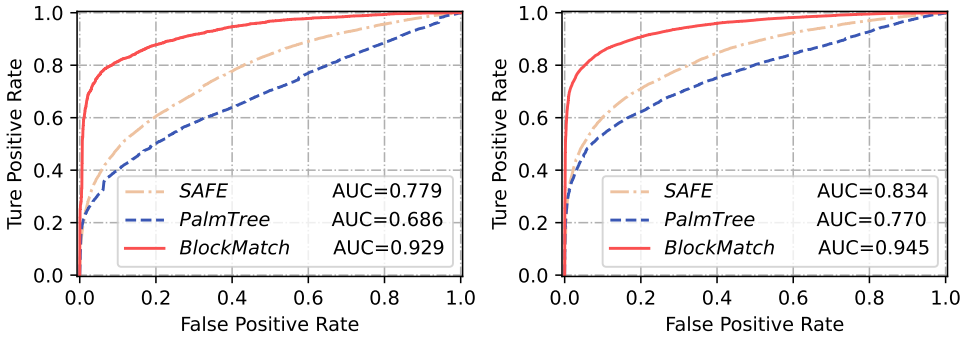
Figure 7. Results of basic block similarity detection across optimizations.



(**a**) goahead (**b**) diffutils



(**c**) openssl (**d**) coreutils



(**e**) sqlite (**f**) linux

Figure 8. Results of basic block similarity detection across compilers.

4.3. One-to-Many Basic Block Matching

To evaluate the performance of `BlockMatch` and other baselines in the last-mile problem of the BCSD tasks regarding matching among numerous basic blocks, we conducted ex-

periments to retrieve the most similar basic block from the large pool-size set of basic blocks. We performed `BlockMatch` and baselines in both cross-compiler and cross-optimization scenarios to provide a comprehensive evaluation of their performance. Moreover, we employed multiple pool sizes to gauge `BlockMatch`'s capability under varying levels of difficulty and to discuss the effect of pool size on BCSD approaches. In order to determine the range of pool size values, we conduct a statistical analysis on the distribution of the number of basic blocks in functions across all projects in the dataset. As shown in Figure 9, functions with more than 10 basic blocks account for 49.3% of the total, while functions with more than 1000 basic blocks only constitute 0.01%. Thus, for our experiment, we select a pool size within or less than 1000.

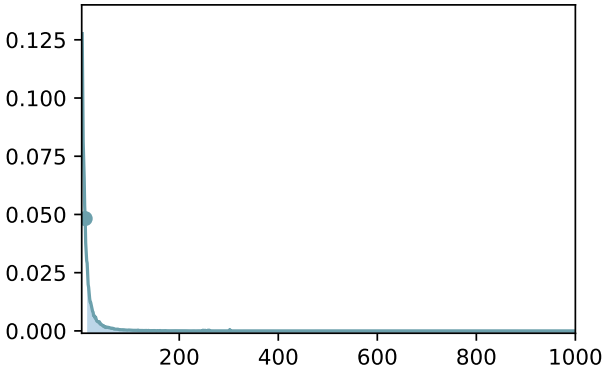


Figure 9. The distribution of the number of basic blocks in functions, where the shaded area represents functions with more than 10 basic blocks, accounts for 49.3% of the total.

Tables 1 and 2 report the recall@1 and MRR results for each approach across different compilers and optimization options in multiple pool sizes (10 and 100). `BlockMatch` significantly surpasses all baselines in terms of recall@1 and MRR. For example, in the cross-optimization (XO) experiment with a pool size of 10, `BlockMatch` yields a recall@1 score of 0.912 and a MRR score of 0.943, increasing the recall@1 by 14.3% and 12.6% as compared to SAFE and PalmTree. In the cross-compile (XC) experiment (pool size = 100), `BlockMatch` outperforms its nearest baseline competitor, PalmTree [23], by 0.407 for recall@1 and by over 40% for the MRR.

Table 1. BCSD results for multiple scenarios at the basic block level (pool size = 10).

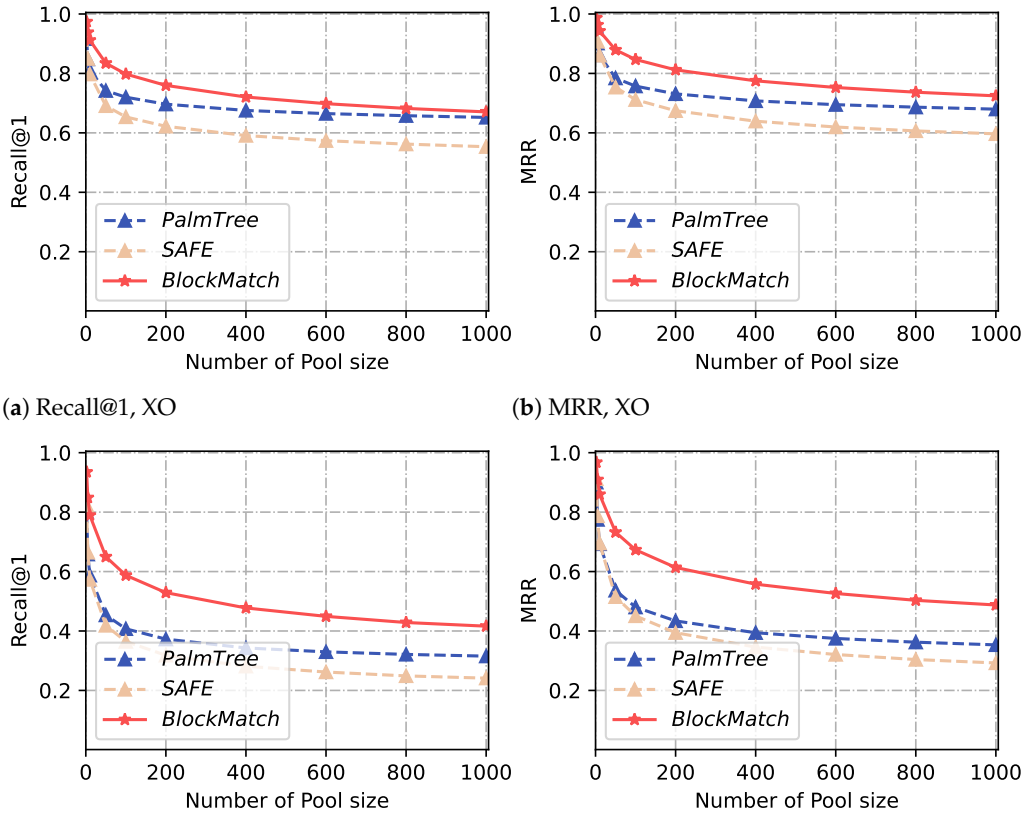
	Recall@1			**MRR**		
Models	**XO**	**XC**	**Average**	**XO**	**XC**	**Average**
SAFE	0.798	0.573	0.685	0.861	0.695	0.778
PalmTree	0.810	0.587	0.699	0.862	0.693	0.778
BlockMatch	0.912	0.790	0.851	0.943	0.860	0.901

Table 2. BCSD results for multiple scenarios at the basic block level (pool size = 100).

	Recall@1			**MRR**		
Models	**XO**	**XC**	**Average**	**XO**	**XC**	**Average**
SAFE	0.654	0.364	0.509	0.712	0.451	0.581
PalmTree	0.720	0.407	0.564	0.757	0.480	0.619
BlockMatch	0.798	0.588	0.693	0.847	0.673	0.760

The experimental results from Tables 1 and 2 reveal a decline in recall@1 and MRR for BCSD approaches as the pool size increases. To delve deeper into the impact of pool size on the performance of BCSD approaches, we conduct experiments with pool sizes ranging between 1 and 1000. Figure 10 displays the result curves of these experiments

according to variations in pool size in cross-optimization and cross-compiler experiment configurations. For ease of observation, we apply a logarithmic x-axis in Figure 10. Across a range of pool sizes from 1 to 1000, `BlockMatch` consistently outperforms PalmTree and SAFE in terms of Recall@1 and MRR. Furthermore, we observe that `BlockMatch` exhibits an more significant advantage over the baselines in cross-compiler experiments. This is attributable to `BlockMatch` using a contrastive training framework to generate a substantial amount of high-quality samples for training, thereby enabling our model to effectively handle hard samples.



(**a**) Recall@1, XO (**b**) MRR, XO

(**c**) Recall@1, XC (**d**) MRR, XC

Figure 10. Results of multiple scenarios with different pool size.

4.4. Ablation Study

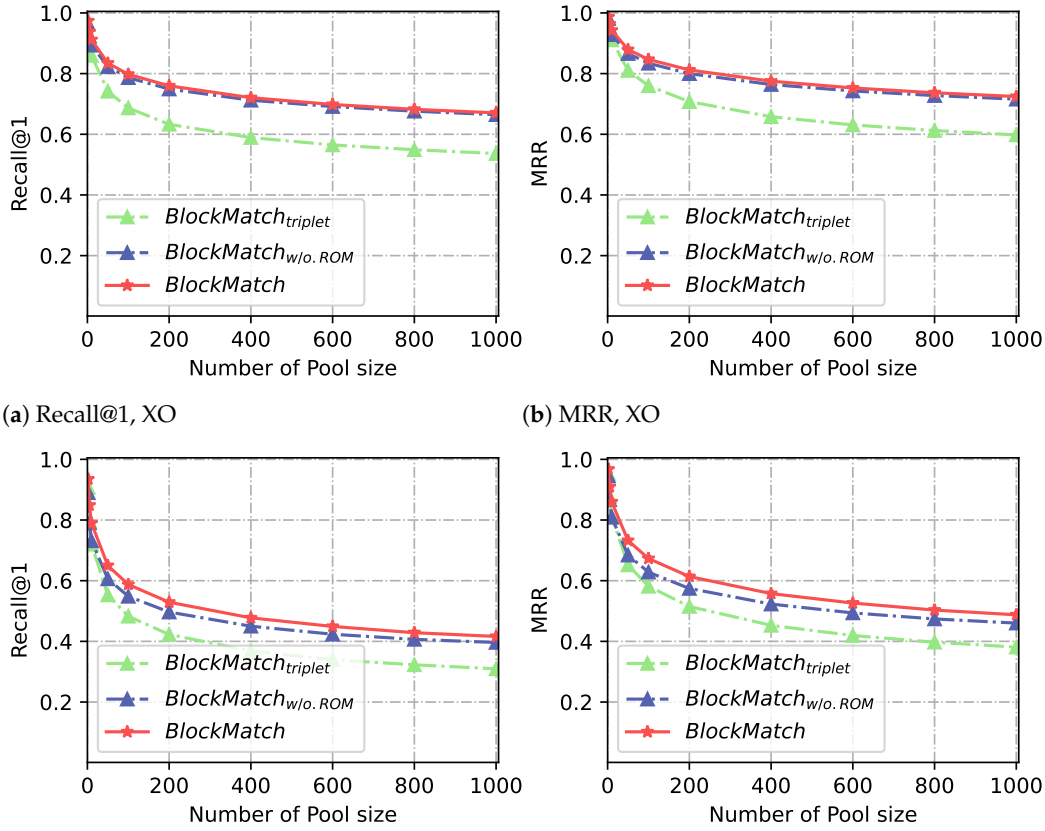
We conduct experiments on `BlockMatch` to evaluate the contributions of the contrastive training framework and root operand model pre-training task. For observation, a detailed examination of the specific configurations of `BlockMatch` variants is provided in Table 3. In the Table, a checked circle (◯) indicates this `BlockMatch` variant uses the corresponding technique, whereas a cross (×) denotes that this `BlockMatch` variant does not use the corresponding technique.

Table 3. The setting of BlockMatch variants.

	BlockMatch	**BlockMatch**$_{triplet}$	**BlockMatch**$_{w/o.\ ROM}$
Contrastive Training Framework	◯	×	◯
Triplet Loss Training	×	◯	×
Root Operand Model	◯	◯	×
Masked Language Model	◯	◯	◯

Contrastive Training Framework. Figure 11 illustrates that `BlockMatch` outperforms `BlockMatch`$_{triplet}$ in basic block similarity detection evaluations across various compilers

and optimizations. For example, when the number of basic blocks in the pool increases to 1000 in the cross-optimization scenario, the recall@1 of $\texttt{BlockMatch}_{triplet}$ drops to 0.537. In contrast, `BlockMatch` maintains a recall@1 of 0.671. This highlights the effectiveness of our contrastive training framework, which utilizes large-batch hard samples for more comprehensive model training. Consequently, `BlockMatch` converges towards optimal parameters and achieves higher performance.



(**a**) Recall@1, XO (**b**) MRR, XO

(**c**) Recall@1, XC (**d**) MRR, XC

Figure 11. Results of the ablation study.

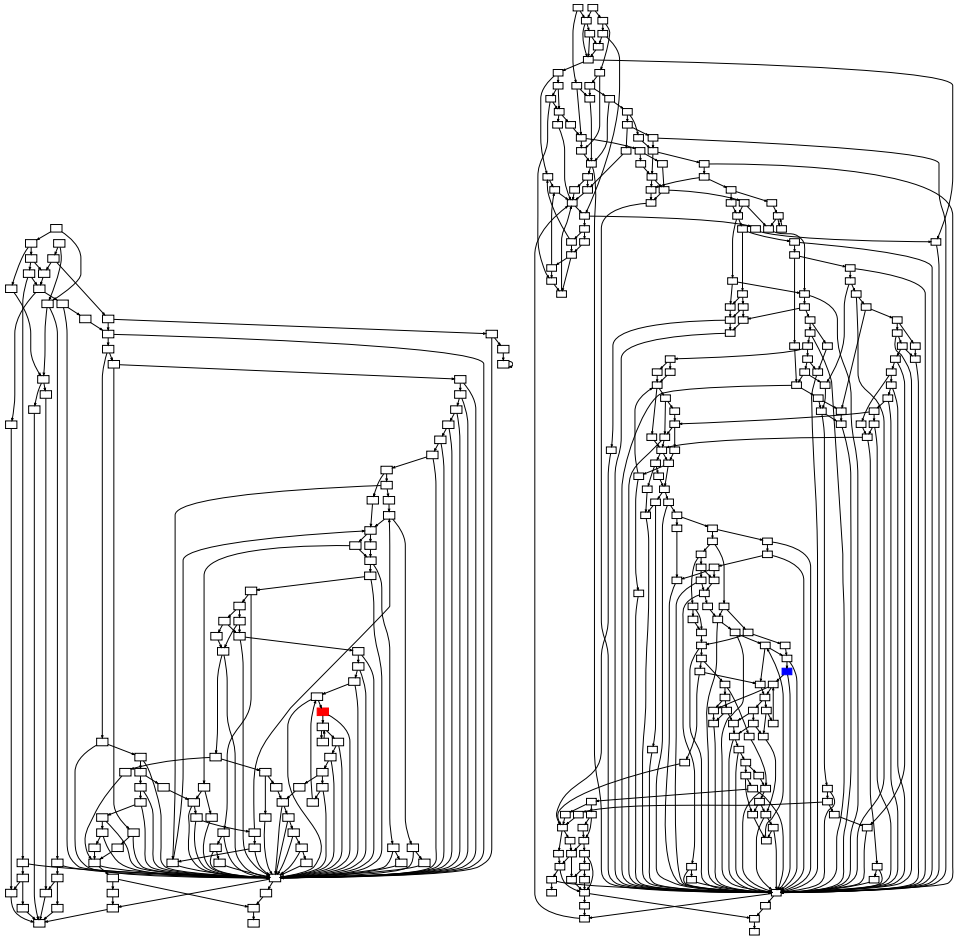
Root Operand Model. The root operand model is used to help our model comprehend the root semantics of unseen operands. The results presented in Figure 11 clearly demonstrate that `BlockMatch` consistently outperforms $\texttt{BlockMatch}_{w/o.ROM}$ in terms of recall@1 and MRR for different pool sizes (ranging from 1 to 1000), particularly in the cross-compiler scenario. These findings suggest that our root operand model pre-training task mitigates the impact of OOV operands on our model, which makes our model perform better on binary code similarity detection.

4.5. Case Study

We conducted a experiment in vulnerability detection to evaluate the effectiveness of `BlockMatch` in real-world situations. The experiment utilized `BlockMatch` to locate the starting position of the vulnerability CVE-2022-0778 in the suspected vulnerable function, aiming to improve the efficiency of security researchers when analyzing vulnerabilities. CVE-2022-0778 is a vulnerability in OpenSSL caused by the improper handling of non-prime moduli in the BN_mod_sqrt function, which can potentially lead to an infinite loop and result in a denial of service (DoS) issue. The affected versions range from 1.1.1 to 1.1.1l.

Figure 12a shows a control flow graph of the confirmed vulnerability function (CVE-2022-0778). According to the symbol tables, we have identified the start position of the vulnerability as the red block. Figure 12b shows the CFGs of a function affected by

vulnerability CVE-2022-0778. Identifying the start position of the vulnerability in a function with hundreds of basic blocks is a time-consuming task, but it is crucial for security researchers to analyze the vulnerability. The existing function-oriented BCSD tools have difficulty in solving this last-mile problem. We use our approach, `BlockMatch`, to quickly and accurately identify the corresponding blocks in the suspected vulnerable function by searching for blocks similar to the red block. This helps save a significant amount of time in vulnerability analysis. We compare our approach with PalmTree and SAFE in this scenario, and Table 4 shows the results, where candidate-k represents the kth candidate basic block, ranked in descending order of similarity. Note that we filter out basic blocks with a similarity lower than 0.9. PalmTree and `BlockMatch` can both identify the start position of the vulnerability at $k = 1$, while SAFE requires $k = 4$ to identify the start position of the vulnerability. This indicates that `BlockMatch` and PalmTree have higher recall compared to SAFE in identifying the start block of vulnerabilities in functions. Additionally, when retrieving the top 5 candidates, `BlockMatch` retrieves one true positive and two false positives, while PalmTree and SAFE retrieve one true positive and four false positives. This suggests that `BlockMatch` has better discrimination ability between similar and dissimilar basic blocks compared to the other two baselines, which can better alleviate the workload of security researchers during vulnerability analysis.



(**a**) BN_mod_sqrt O0 (**b**) BN_mod_sqrt obfuscated

Figure 12. On the left is the control flow graph of the confirmed vulnerable function (CVE-2022-0778), with the red block indicating the start position of the vulnerability. On the right is the control flow graph from the suspected vulnerable function. Through our manual confirmation, the blue block represents the start position of the vulnerability.

Table 4. The result of locating the start position of vulnerability CVE-2022-0778 in the suspicious function's basic blocks.

	Candidate-1	Candidate-2	Candidate-3	Candidate-4	Candidate-5
BlockMatch	◯	×	×	-	-
PalmTree	◯	×	×	×	×
SAFE	×	×	×	◯	×

5. Discussion

In this paper, we propose a novel fine-grained BCSD approach, named `BlockMatch`, using NLP techniques for basic block matching. The experiments show that `BlockMatch` outperforms the state-of-the-art methods PalmTree and SAFE. Here, we discuss the following questions:

Why does BlockMatch achieve good performance in basic block matching? The basic block matching in BCSD tasks belongs to fine-grained tasks. Compared with function-level BCSD methods, it specifically identifies which basic blocks are similar in binary files, while also facing a larger number of comparison examples. Experimental results have shown that the contrastive training framework plays a significant role in achieving high performance for `BlockMatch`. The contrastive training framework leverages DWARF debug information to automatically generate a large number of high-quality contrastive samples, facilitating the fine-tuning of `BlockMatch`, which enables `BlockMatch` to converge faster and handle hard samples effectively. Table 5 shows the average results of `BlockMatch` and other baselines, which demonstrates that `BlockMatch` generally outperforms PalmTree and SAFE in basic block similarity detection. Comparative results against other baselines demonstrate that this improvement is particularly pronounced in cross-compiler scenarios. For example, in the cross-compilers experiment (pool size = 10), `BlockMatch` achieves a recall@ of 0.790, while SAFE and PalmTree achieve recalls of 0.573 and 0.587, respectively.

Table 5. The Average result of BlockMatch and other baselines.

Approaches	Pool Size = 10		Pool Size = 100	
	Recall@1	MRR	Recall@1	MRR
BlockMatch	0.851	0.901	0.693	0.760
PalmTree	0.699	0.778	0.564	0.619
SAFE	0.685	0.778	0.509	0.581

Why does `BlockMatch` show a slight decrease in performance when used for cross-compiler basic block matching? Based on the results obtained from one-to-one and one-to-many basic block matching, it is evident that `BlockMatch` demonstrates slightly lower performance in cross-compiler basic block matching compared to cross-optimization basic block matching. Actually, the same issue also occurs in other approaches like SAFE and PalmTree. Analyzing this issue will help us make further improvements to our approaches in the future. We compare binary snippets compiled by Clang and GCC using -O0 option without any compiler optimization intervention and differences in the preferred instructions they generate. For example, in binary files compiled with GCC, there are numerous *endbr64* instructions. These instructions are part of Intel's Control-Flow Enforcement Technology, which provides protection against Return-oriented Programming (ROP) attacks. However, in binary files compiled with Clang, the frequency of the *endbr64* instruction is much lower. Similar issues have also been mentioned in other literature [58]. When the differences exceed the tolerance threshold of the model, it leads to the misidentification of similar basic blocks by the model. Although using a contrastive training framework can partially mitigate this issue, it is still challenging to completely eliminate the impact of these differences. We notice that the control flow graphs remain relatively consistent across different compilers, so we can consider using graph node alignment based on control flow graphs in future work to help overcome this issue.

Can BlockMatch work for binary files from other architectures? In this paper, we primarily focus on evaluating the performance of `BlockMatch` and other baselines on the `x86` architecture. Since our approach is architecture-agnostic, by training with binary files from other architectures, it can also be applied to other architectures. This is also a direction for our future research.

6. Conclusions

In this paper, we propose a novel fine-grained approach based on NLP techniques and contrastive learning for basic block matching to solve the last-mile problem in BCSD tasks. To mitigate bias caused by partial samples, we propose a contrastive training framework to generate high-quality training samples and perform large batch sizes using NT-Xent loss for model training. To mitigate the impact of OOV operands, we employ a root operand model pre-training task to help our model learn root semantics for unseen operands. We implemented the prototype of `BlockMatch`. Our experimental results show that `BlockMatch` surpasses the cutting-edge approaches SAFE and PalmTree in binary code similarity detection at the basic block level. The ablation study shows our proposed root operand model and contrastive training framework have positive effects for `BlockMatch`.

Author Contributions: Methodology, Z.L. and P.W.; software, Z.L.; formal analysis, Z.L., P.W., W.X. and B.W.; data curation, Z.L.; writing—original draft preparation, Z.L.; writing—review and editing, Z.L., P.W., W.X., X.Z. and B.W.; supervision, B.W.; project administration, B.W.; funding acquisition, P.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National University of Defense Technology Research Project (ZK20-17, ZK20-09), the National Natural Science Foundation China (62272472, 61902405), the HUNAN Province Natural Science Foundation (2021JJ40692), and the National Key Research and Development Program of China under Grant No. 2021YFB0300101.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BCSD	Binary Code Similarity Detection
ML	Machine Learning
CFG	Control Flow Graph
GNN	Graph Neural Network
NLP	Natural language processing
MLM	Masked Language Model
ROM	Root Operand Model
OOV	Out-of-Vocabulary
BSM	Block Similarity Model
DWARF	Debugging with Attributed Record Formats
ROP	Return-oriented Programming

References

1. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1157–1177. [CrossRef]
2. Ming, J.; Xu, D.; Jiang, Y.; Wu, D. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 253–270.

3. Cesare, S.; Xiang, Y.; Zhou, W. Control flow-based malware variantdetection. *IEEE Trans. Dependable Secur. Comput.* **2013**, *11*, 307–317. [CrossRef]
4. Hu, X.; Chiueh, T.C.; Shin, K.G. Large-scale malware indexing using function-call graphs. In Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 9–13 November 2009; pp. 611–620.
5. Bayer, U.; Comparetti, P.M.; Hlauschek, C.; Kruegel, C.; Kirda, E. Scalable, behavior-based malware clustering. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 8–11 February 2009; Volume 9, pp. 8–11.
6. Kruegel, C.; Kirda, E.; Mutz, D.; Robertson, W.; Vigna, G. Polymorphic worm detection using structural information of executables. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Seattle, WA, USA, 7–9 September 2005; pp. 207–226.
7. Liu, B.; Li, W.; Huo, W.; Li, F.; Zou, W.; Zhang, C.; Piao, A. αDiff: Cross-version binary code similarity detection with DNN. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 667–678. [CrossRef]
8. Feng, Q.; Wang, M.; Zhang, M.; Zhou, R.; Henderson, A.; Yin, H. Extracting conditional formulas for cross-platform bug search. In Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 346–359. [CrossRef]
9. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724.
10. Luo, Z.; Wang, P.; Wang, B.; Tang, Y.; Xie, W.; Zhou, X.; Liu, D.; Lu, K. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, USA, 27 February–3 March 2023.
11. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 896–899. [CrossRef]
12. David, Y.; Yahav, E. Tracelet-based code search in executables. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, CA, USA, 13–17 June 2014; pp. 349–360. [CrossRef]
13. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jTrans: Jump-Aware Transformer for Binary Code Similarity. *arXiv* **2022**, arXiv:2205.12713.
14. Lin, J.; Wang, D.; Chang, R.; Wu, L.; Zhou, Y.; Ren, K. EnBinDiff: Identifying Data-only Patches for Binaries. *IEEE Trans. Dependable Secur. Comput.* **2021**, 20, *343–359*. [CrossRef]
15. Ding, S.H.; Fung, B.C.; Charland, P. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 19–23 May 2019; pp. 472–489. [CrossRef]
16. David, Y.; Partush, N.; Yahav, E. Statistical similarity of binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA, USA, 13–17 June 2016; pp. 266–280.
17. Yang, S.; Cheng, L.; Zeng, Y.; Lang, Z.; Zhu, H.; Shi, Z. Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, 21–24 June 2021; pp. 224–236.
18. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Kuan, T.H.B. BinGo: Cross-Architecture cross-os binary search. In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 678–689. [CrossRef]
19. David, Y.; Partush, N.; Yahav, E. Similarity of binaries through re-optimization. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain, 18–23 June 2017; pp. 79–94.
20. Zynamics. BinDiff. 2021. Available online: https://www.zynamics.com/bindiff.html (accessed on 24 May 2023).
21. Gao, D.; Reiter, M.K.; Song, D. Binhunt: Automatically finding semantic differences in binary programs. In Proceedings of the International Conference on Information and Communications Security, Birmingham, UK, 20–22 October 2008; pp. 238–255.
22. Pewny, J.; Schuster, F.; Bernhard, L.; Holz, T.; Rossow, C. Leveraging semantic signatures for bug search in binary programs. In Proceedings of the 30th Annual Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014; pp. 406–415.
23. Li, X.; Yu, Q.; Yin, H. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 15–19 November 2021; pp. 3236–3251.
24. Pei, K.; Xuan, Z.; Yang, J.; Jana, S.; Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv* **2020**, arXiv:2012.08680.
25. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Baldoni, R.; Querzoni, L. Safe: Self-attentive function embeddings for binary similarity. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Gothenburg, Sweden, 19–20 June 2019; pp. 309–329.
26. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable Graph-based Bug Search for Firmware Images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security—CCS'16, Vienna, Austria, 24–28 October 2016; ACM Press: New York, NY, USA, 2016; pp. 480–491. [CrossRef]

27. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. 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—CCS'17, Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376. [CrossRef]
28. Schroff, F.; Kalenichenko, D.; Philbin, J. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 815–823.
29. Marcelli, A.; Graziano, M.; Ugarte-Pedrero, X.; Fratantonio, Y.; Mansouri, M.; Balzarotti, D. How Machine Learning is Solving the Binary Function Similarity Problem. In Proceedings of the Usenix Security 2022, Boston, MA, USA, 10–12 August 2022; pp. 83–101.
30. Wu, Z.; Xiong, Y.; Yu, S.X.; Lin, D. Unsupervised feature learning via non-parametric instance discrimination. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 3733–3742.
31. Chen, T.; Kornblith, S.; Norouzi, M.; Hinton, G. A simple framework for contrastive learning of visual representations. In Proceedings of the International Conference on Machine Learning (PMLR), Virtual, 13–18 July 2020; pp. 1597–1607.
32. Farhadi, M.R.; Fung, B.C.; Charland, P.; Debbabi, M. Binclone: Detecting code clones in malware. In Proceedings of the 2014 Eighth International Conference on Software Security and Reliability (SERE), San Francisco, CA, USA, 30 June–2 July 2014; pp. 78–87.
33. Ding, S.H.; Fung, B.C.; Charland, P. Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining—KDD'16, San Francisco, CA, USA, 13–17 August 2016; ACM Press: New York, NY, USA, 2016; pp. 461–470. [CrossRef]
34. Kim, G.; Hong, S.; Franz, M.; Song, D. Improving cross-platform binary analysis using representation learning via graph alignment. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2022; pp. 151–163.
35. Yang, S.; Dong, C.; Xiao, Y.; Cheng, Y.; Shi, Z.; Li, Z.; Sun, L. Asteria-Pro: Enhancing Deep-Learning Based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 1–40. [CrossRef]
36. Qasem, A.; Debbabi, M.; Lebel, B.; Kassouf, M. Binary Function Clone Search in the Presence of Code Obfuscation and Optimization over Multi-CPU Architectures. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, Melbourne, Australia, 10–14 July 2023; pp. 443–456.
37. Ahn, S.; Ahn, S.; Koo, H.; Paek, Y. Practical binary code similarity detection with bert-based transferable similarity learning. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 4–8 December 2022; pp. 361–374.
38. Myles, G.; Collberg, C. K-gram based software birthmarks. In Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, NM, USA, 13–17 March 2005; pp. 314–318.
39. Jang, J.; Woo, M.; Brumley, D. Towards automatic software lineage inference. In Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, DC, USA, 14–16 August 2013; pp. 81–96.
40. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the 2016 Network and Distributed System Security Symposium, Internet Society, Reston, VA, USA, 21–24 February 2016; pp. 21–24. [CrossRef]
41. Duan, Y.; Li, X.; Wang, J.; Yin, H. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In Proceedings of the 27rd Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 23–26 February 2020. [CrossRef]
42. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1145–1152. [CrossRef]
43. Le, Q.; Mikolov, T. Distributed representations of sentences and documents. In Proceedings of the International Conference on Machine Learning. PMLR, Beijing, China, 22–24 June 2014; pp. 1188–1196.
44. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. 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 Technologiese, Minneapolis, MN, USA, 2–7 June 2019; Volume 1, pp. 4171–4186.
45. He, P.; Liu, X.; Gao, J.; Chen, W. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv* **2020**, arXiv:2006.03654.
46. Rays, H. IDA Pro. 2021. Available online: https://www.hex-rays.com/products/ida/ (accessed on 21 December 2021).
47. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv* **2019**, arXiv:1907.11692.
48. Committee, D.S. The DWARF Debugging Standard. 2021. Available online: https://dwarfstd.org/ (accessed on 21 December 2021).
49. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2019.
50. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-Art Natural Language Processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Association for Computational Linguistics, Virtual, 16–18 November 2020; pp. 38–45.

51. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed Representations of Words and Phrases and their Compositionality. *arXiv* **2013**, arXiv:1310.4546.
52. Embedthis. GoAHead. 2023. Available online: https://www.embedthis.com/goahead/ (accessed on 20 May 2023).
53. GNU. Coreutils—GNU Core Utilities. 2023. Available online: https://www.gnu.org/software/coreutils/coreutils.html (accessed on 19 May 2023).
54. OpenSSL. OpenSSL. 2023. Available online: https://www.openssl.org/ (accessed on 19 May 2023).
55. GNU. Diffutils—GNU Core Utilities. 2023. Available online: https://www.gnu.org/software/diffutils/diffutils.html (accessed on 19 May 2023).
56. Organization, L.K. The Linux Kernel Archives. 2023. Available online: https://www.kernel.org/ (accessed on 22 May 2023).
57. Consortium, S. SQLite. 2023. Available online: https://www.sqlite.org/index.html (accessed on 21 May 2023).
58. Xu, X.; Feng, S.; Ye, Y.; Shen, G.; Su, Z.; Cheng, S.; Tao, G.; Shi, Q.; Zhang, Z.; Zhang, X. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023; pp. 1106–1118.