BinSimDB: Benchmark Dataset Construction for *Fine-Grained* Binary Code Similarity Analysis

Fei Zuo[1(✉)], Cody Tompkins[1], Qiang Zeng[2], Lannan Luo[2], Yung Ryn Choe[3], and Junghwan Rhee[1]

[1] University of Central Oklahoma, Edmond, OK 73034, USA
{fzuo,ctompkins6,jrhee2}@uco.edu
[2] George Mason University, Fairfax, VA 22030, USA
{zeng,lluo4}@gmu.edu
[3] Sandia National Laboratories, Livermore, CA 94551, USA
yrchoe@sandia.gov

Abstract. Binary Code Similarity Analysis (BCSA) has a wide spectrum of applications, including plagiarism detection, vulnerability discovery, and malware analysis, thus drawing significant attention from the security community. However, conventional techniques often face challenges in balancing both accuracy and scalability simultaneously. To overcome these existing problems, a surge of deep learning-based work has been recently proposed. Unfortunately, many researchers still find it extremely difficult to conduct relevant studies or extend existing approaches. First, prior work typically relies on proprietary benchmark without making the entire dataset publicly accessible. Consequently, a large-scale, well-labelled dataset for binary code similarity analysis remains precious and scarce. Moreover, previous work has primarily focused on comparing at the function level, rather than exploring other finer granularities. Therefore, we argue that the lack of a fine-grained dataset for BCSA leaves a critical gap in current research. To address these challenges, we construct a benchmark dataset for fine-grained binary code similarity analysis called BinSimDB, which contains equivalent pairs of smaller binary code snippets, such as basic blocks. Specifically, we propose BMerge and BPair algorithms to bridge the discrepancies between two binary code snippets caused by different optimization levels or platforms. Furthermore, we empirically study the properties of our dataset and evaluate its effectiveness for the BCSA research. The experimental results demonstrate that BinSimDB significantly improves the performance of binary code similarity comparison.

Keywords: Binary analysis · Benchmark dataset · Binary code similarity analysis.

1 Introduction

The aim of binary code similarity analysis (BCSA) is to determine whether two binary code snippets are equivalent or not. A technique for BCSA can be

applied to many security-related applications. For example, we take the cross-platform vulnerability search as a motivation problem. Given a patched binary code snippet from one platform, by BCSA we can identify whether there exists a vulnerability due to code reuse in the target binary coming from a different platform. However, since the binary code snippets can be generated by using distinct optimization levels or targeting different platforms, the two pieces of code that need to be compared are usually too different to be matched. To address this challenge, a surge of recent work shifts the focus to the popular arsenal of deep learning. For example, graph-based learning methods [10, 15, 34, 37] extract graph information from binary code and use them as the basis for similarity detection because a binary executable is intrinsically associated with some graph representations such as control flow graphs (CFGs). Other methods [2,9,17,20,31] consider the assembly code as a language, thus developing NLP-inspired techniques to detect similarity.

However, high-quality datasets play a significant role in any data-driven application. Thus, the construction of datasets for learning-based security studies has raised awareness among prior researchers. They believe that the unavailability of data is a widespread obstacle for security research [25,28,32]. This problem is particularly severe in the field of BCSA. For example, after investigating 43 papers in this area, Kim et al. [14] found "*only two of them opened their entire dataset, which makes it difficult to reproduce or extend previous work*".

Moreover, although a few previous researchers have started making their dataset publicly available for the convenience of subsequent work, they primarily focus on the similarity detection at the function level rather than other finer granularities. Nevertheless, the function-level comparison is less useful for the applications where subtle discrepancies matter. For example, when patching a piece of vulnerable code, it is usually unnecessary to rewrite the entire function. Therefore, we believe the unavailability of a fine-grained dataset for BCSA requires sufficient attention but has not yet been well resolved. It is worth noting that, as the first deep learning-based approach to detecting binary code similarity at the basic block level, `InnerEye` [38] proposed a dataset construction method. However, their method is limited by two drawbacks.

First, `InnerEye` extracts binary code directly from the backend of a specific compiler, namely LLVM. However, the binary code obtained from the reverse engineering tool is not exactly identical to the binary code generated by the compiler, which creates a gap in practical application. Second, when matching two equivalent basic block pairs, `InnerEye` relies on the *annotations* generated by the LLVM facility to annotate every basic block. According to the official manual [27], the formatted string used by LLVM to identify a basic block and its parent function is "*hopefully unique*" but there is no guarantee. Hence, the matching result of equivalent basic blocks is not exactly accurate. Especially when compiling the source code using different optimization levels, it is highly possible that basic blocks from a lower optimization level cannot successfully find an exact match in a new binary obtained by using a higher optimization

level. `InnerEye` suffers from a failure to address this case, thus raising concerns about the data quality.

Generating basic block-level equivalent pairs is never a trivial task. Previous studies [2, 14, 17, 20] have focused on the similarity between functions, allowing them to establish ground truth using function names. However, unlike functions, basic blocks lack an off-the-shelf unique identifier. Therefore, we propose using source code information to annotate every basic block. Still, establishing a one-to-one mapping among basic blocks across different optimization levels or platforms remains challenging even with source code information. The two main challenges ahead are: ❶ A single line of source code may correspond to multiple basic blocks, and the number of basic blocks originating from the same line of source code may vary across different architectures; ❷ Compiling source code with a higher optimization level may lead to the merging or reorganization of original basic blocks generated at a lower optimization level due to compiler optimization behavior. To address these challenges in dataset construction, we have developed BMerge and BPair algorithm. We will shed light on our observations using a concrete motivation example and also the proposed solutions in Section 3.

The key contributions of our work include:

- We construct a fine-grained dataset BinSimDB[1], which consists of 4,426,258 equivalent assembly code pairs, for facilitating BCSA studies. To the best of our knowledge, we are the first to release a diverse set of fine-grained equivalent binary code pairs at this scale.
- Not only the comprehensive benchmark for BCSA, we also make our automated scripts publicly accessible, so that future academics are able to easily reproduce or extend the dataset for various research purposes.
- We construct the reliable ground truth by adopting source code information. In particular, we propose BMerge and BPair algorithm to handle the issues raised in the equivalent binary code matching at a fine granularity.
- We empirically investigate the properties of BinSimDB, and evaluate its effectiveness for the BCSA research. The experimental results demonstrate that our dataset can greatly help to improve the performance of binary code similarity comparison.

2 Background

2.1 Binary Code Similarity Analysis

Binary code similarity analysis (BCSA) is the process of comparing two or more binary code snippets to determine how similar they are. The goal of this analysis is to identify potential code reuse or plagiarism between different software programs, as well as to discover potential security vulnerabilities or malware. Note that in this paper, we will use the terms 'binary code' and 'assembly code' interchangeably, unless otherwise specified.

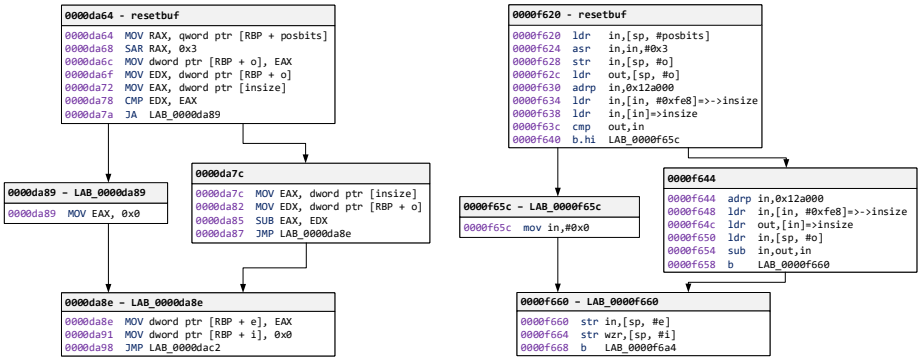
[1] We share our dataset with the cybersecurity community in the following link: `https://uco-cyber.github.io/research/#binsimdb`.

```
diff --git a/unlzw.c b/unlzw.c
index fb9ff76..8f8cbee 100644
--- a/unlzw.c
+++ b/unlzw.c
@@ -240,7 +240,8 @@ int unlzw(in, out)
         int   o;

      resetbuf:
-        e = insize-(o = (posbits>>3));
+        o = posbits >> 3;
+        e = o <= insize ? insize - o : 0;

         for (i = 0 ; i < e ; ++i) {
             inbuf[i] = inbuf[i+o];
```

Fig. 1: A patch in `gzip` for CVE-2010-0001.



(a) Binary code in x86-64 (b) Binary code in AArch64

Fig. 2: A pair of equivalent binary code snippets from the same source code.

We regard two binary code snippets as similar if they come from the same piece of source code. For example, Figure 1 shows a source code patch for CVE-2010-0001. In detail, there is an integer underflow vulnerability in the `unlzw` function of `gzip` before 1.4 on 64-bit platforms, thus possibly causing a denial of service or allowing remote attackers to execute arbitrary code. We compiled the above patch code for two different platforms, x86-64 and AArch64, resulting in two equivalent binary code snippets, as shown in Figure 2(a) and Figure 2(b), respectively. For the sake of presentation, we particularly use the optimization level `O0` when compiling the patched code. Though control flow graphs (CFGs) across the two platforms remain similar, the binary code looks quite different because of distinct instruction set architectures (ISAs) and registers. Furthermore, there are 75 basic blocks in the `unlzw` function overall, while code changes caused by the patch only take up approximately 5.3% of the entire function. Hence, this example showcases that considering fine-grained discrepancy in some practical applications is of great necessity as well.

2.2 Toolchain

It is noteworthy that the third-party applications utilized in this study are non-proprietary, which provides sufficient flexibility for the academic community to re-use all the research resources. In detail, we develop our system using Python in Ubuntu 22.04. For the software reverse engineering tool, we choose the free and open sourced framework `Ghidra`[2], which is developed by the National Security Agency (NSA) of the United States. `Ghidra` is capable of analyzing compiled code on any platform, whether Linux, Windows, or macOS. On top of that, `Ghidra` also enables users to perform automated analysis with scripts in Python via `Ghidrathon` extension. Today, `Ghidra` has been widely used by the security community [23] and is also regarded as well-matched in strength with its expensive competitor `IDA Pro`[3] [30].

Furthermore, the GNU binary utility `addr2line`[4] is adopted to translate hexadecimal addresses in a executable into source code file names and line numbers. Given an address in an executable or an offset in a section of a relocatable object, it uses the debugging information to figure out which file name and line number, in the source code, are associated with it. By this means, we can annotate every basic block using source code information. As a result, basic block splitting and combination, or function inline will not be a problem for building a self-evident connection between two assembly code snippets even if they are from different architectures or optimization levels. Besides, `llvm-addr2line` developed by the LLVM project can be used as a drop-in replacement for GNU's `addr2line`. The two utilities are interchangeable in this project.

3 Methodology

3.1 Key Observations

To construct the ground truth using source code information, we traverse all addresses for a given basic block i and leverage `addr2line` to obtain a label set $\mathcal{A}_i$, where each element corresponds to a source file and line number associated with the basic block. For example, we annotate every basic block in Figure 2(b), and the result is shown in Figure 3. However, we observe that *a single line of source code may correspond to multiple basic blocks*. Consequently, there may be two different basic blocks i and j in the same function, such that $\mathcal{A}_i \cap \mathcal{A}_j \neq \emptyset$. This inevitably leads to confusion when pairing two basic blocks across different optimization levels or platforms. To ensure the uniqueness of the annotation for a binary code unit under a given setting, we propose using the BMerge Algorithm (shown in Algorithm 1) to handle those overlapping basic blocks.

Our second observation is, due to the nature of compilers, *multiple basic blocks may be recombined or spread across other basic blocks* when using higher

[2] `https://ghidra-sre.org/`
[3] `https://hex-rays.com/ida-pro/`
[4] `https://www.gnu.org/software/binutils/`

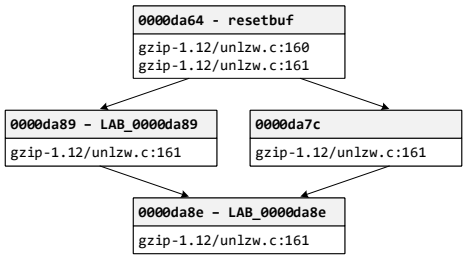


Fig. 3: Basic block annotation with corresponding source file and line numbers.

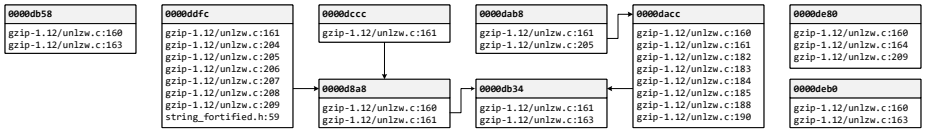


Fig. 4: Binary code in AArch64 obtained by using `O3` as the optimization level.

optimization levels. For example, when compiling the patched code in Figure 2(a) using `O3` on the AArch64 platform, no specific basic blocks obtained are purely derived from the patched code. Instead, the original four basic blocks in Figure 2(c) are recombined and spread across another nine basic blocks, as shown in Figure 4. We can also see some isolated basic blocks. Note that they need to connect to the CFG through some other basic blocks rather than being directly linked. The basic blocks used to connect all the nodes in Figure 4 are not depicted if they are entirely generated from source code beyond the patch. This phenomenon introduces ambiguity when paring two basic blocks across different optimization levels. To this end, we propose the BPair Algorithm (as shown in Algorithm 2) to match equivalent assembly code units.

3.2 Dataset Construction Approach Details

Figure 5 illustrates the construction method of BinSimDB, where all the five steps can be automated by scripts. First, we compile all the binaries with debugging information using the `-g` option. Specifically, source files are complied with two representative compilers (GCC and Clang) across various optimization levels (`O0`, `O1`, `O2`, and `O3`). Four popular ISAs are considered, including x86, x86-64, ARM32, and AArch64. Users can easily extend the script through configuring it to cover additional ISAs.

In the second step, we utilize the facilities provided by `Ghidra` to analyze the compiled binaries. This allows us to collect static analysis results such as binary functions, basic blocks, control flow graphs (CFGs), and call graphs. Notably, we preliminarily sanitize the dataset to exclude external functions that lack actual function bodies. However, for functions from widely-used third-party libraries such as `glibc`, we maintain a dictionary $\mathcal{D}$ to record them. Modern reverse

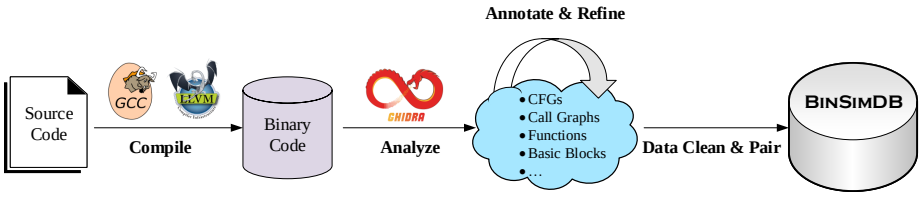


Fig. 5: System overview.

Algorithm 1: BMerge Algorithm

Input: A set of basic blocks, denoted by $\mathcal{S}$, wherein a basic block i is labeled with a set $\mathcal{A}_i$, and $addr_i$ represents the address of i.
Output: The refined basic blocks set $\mathcal{S}$.

```
1  foreach i, j ∈ S, where i ≠ j, do
2      if A_i = A_j then
3          if addr_i < addr_j then
4              Update i by merging j into i
5              S ← S − {j}
6          else
7              Update j by merging i into j
8              S ← S − {i}
9      else if A_j ⊂ A_i then
10         Update i by merging j into i
11         S ← S − {j}
12     else if A_i ⊂ A_j then
13         Update j by merging i into j
14         S ← S − {i}
15 return S
```

engineering applications like `Ghidra` and `IDA Pro` have developed techniques for identifying library functions, so we do not focus on this aspect in our work.

Next, we utilize `addr2line` to annotate every basic block with the corresponding information of source files and line numbers. Specifically, functions generated by compilers rather than the application itself are discarded. This can be easily achieved by referencing the source code information obtained from `addr2line`. More concretely, we discard a basic block i if $\mathcal{A}_i = \emptyset$, where $\mathcal{A}_i$ is the label set consisting of source file names and line numbers. After that, we further refine our dataset via the BMerge Algorithm (as shown in Algorithm 1). For any two basic blocks i and j, they need to be merged into one block as long as their label sets $\mathcal{A}_i$ and $\mathcal{A}_j$ can fulfill any following condition: ❶$\mathcal{A}_i$ and $\mathcal{A}_j$ are identical (Line 2∼8); ❷$\mathcal{A}_j$ is the subset of $\mathcal{A}_i$ (Line 9∼11); ❸$\mathcal{A}_i$ is the subset of $\mathcal{A}_j$ (Line 12∼14). For example, by applying this algorithm, the four basic blocks shown in Figure 2(b) can be integrated into a new block. This resulting

block exactly reflects the patched code in Figure 2 and does not overlap with any other basic blocks. The BMerge Algorithm does not need to be applied in the following two possible cases: 1) there is only one basic block in a given function; 2) every basic block is obtained from a few lines of source code that are not simultaneously used to generate any other basic blocks.

To control the vocabulary size and preserve the semantics of instructions, many previous studies [10,11,18,38] perform normalization on instructions. Similarly, we preprocess the instructions in the dataset according to the following empirical rules: ❶ For numeric constants, according to the sign of the value, we replace a numeric constant with `<POSITIVE>`, `<NEGATIVE>` or `<ZERO>`; ❷ For function calls, if the library function can be identified (i.e. the function name is collected by the dictionary $\mathcal{D}$), we preserve the instruction as its original form. Otherwise, the function names are uniformly replaced with `<FOO>`; ❸ The memory addresses, such as the local destination of a jump instruction, are replaced by `<ADDRESS>`; ❹ Finally, we substitute the token `<STRING>` for all string literals.

Algorithm 2: BPair Algorithm

Input: Two sets of basic blocks, denoted by $\mathcal{U}$ and $\mathcal{V}$, where a basic block i is labeled with a set $\mathcal{A}_i$, and $addr_i$ represents the address of i.
Output: A set $\mathcal{M}$ consisting of equivalent basic block pairs.

Function `Merge`(p, q)
  if $addr_p < addr_q$ **then**
    Update p by merging q into p
    return p
  else
    Update q by merging p into q
    return q
Initialize a bipartite graph $\mathcal{G} = (\mathcal{U}, \mathcal{V}, \mathcal{E})$, where $\mathcal{E} = \emptyset$
foreach $u \in \mathcal{U}, v \in \mathcal{V}$, **do**
  if $\mathcal{A}_u \cap \mathcal{A}_v \neq \emptyset$ **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle u, v\rangle\}$
foreach *connected sub-graph* $\mathcal{C} \subset \mathcal{G}$ **do**
  Pick any basic block i from $\mathcal{C}$, where $i \in \mathcal{U}$
  Pick any basic block j from $\mathcal{C}$, where $j \in \mathcal{V}$
  foreach $k \in \mathcal{C} - \{i, j\}$ **do**
    if $k \in \mathcal{U}$ **then**
      $i \leftarrow$ `Merge` (k, i)
    else if $k \in \mathcal{V}$ **then**
      $j \leftarrow$ `Merge` (k, j)
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i, j)\}$
return $\mathcal{M}$

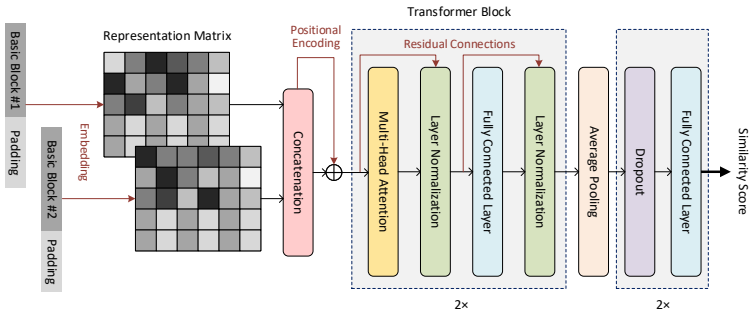


Fig. 6: Architecture of the proposed binary code similarity detector.

At the final stage, we generate equivalent pairs by adopting the BPair Algorithm (as shown in Algorithm 2). The intuition behind this algorithm is to transform the problem of matching equivalent basic blocks into a graph problem. First, the source code of a function can be compiled under different platforms or using different optimization levels, resulting in two sets of basic blocks denoted by $\mathcal{U}$ and $\mathcal{V}$. Based on this, we can build a disconnected bipartite graph $\mathcal{G}$ with partitions $\mathcal{U}$ and $\mathcal{V}$ (Line 8), where the basic blocks $u \in \mathcal{U}$, $v \in \mathcal{V}$ are considered as vertices. Then, we create edges continuously to connect vertices (i.e. basic blocks) in $\mathcal{U}$ to those in $\mathcal{V}$, according to the ground truth regarding basic blocks (Line 9∼11). In other words, for basic blocks $u \in \mathcal{U}$ and $v \in \mathcal{V}$ that are from two different architectures or optimization levels, if their source code overlaps, i.e., the intersection of $\mathcal{A}_u$ and $\mathcal{A}_v$ is not empty (Line 10), then an edge is established between the two vertices u and v (Line 11). After traversing every connected sub-graph of $\mathcal{G}$ based on a disjoint set, we can generate a set of equivalent pairs $\mathcal{M}$ (Line 12∼20). Additionally, all the duplicate pairs are removed from this set to maintain the dataset quality.

3.3 Similarity Detection Model

The success of OpenAI's ChatGPT [5] has sparked significant interest in the academic and industry sectors regarding Large Language Models (LLMs). Herein, the transformer architecture plays a crucial role [29], thus was considered as the fundamental building blocks of LLMs. To demonstrate the benefits of BinSimDB in supporting the future BCSA research, we introduce a Transformer-based binary code similarity detector, as depicted in Figure 6.

At first, a pair of assembly code snippets under different architectures are represented by two embedding matrices. The concatenated input of the two matrices and the corresponding positional encoding are provided to a Transformer-based model. The position encoding aims to capture the order information of each instruction within a code snippet. These generated positional embeddings are added to the concatenated matrix, then sent together to the subsequent Transformer blocks. The embedding input, in the form of three learnable weight

matrices including queries $\mathbf{Q}$, keys $\mathbf{K}$ of dimension d_k, and values $\mathbf{V}$ of dimension d_v, is passed through a scaled dot-product attention. Formula (1) clearly describes the performance of an attention model.

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{1}$$

Then, the results are concatenated through a multi-head attention, where each result of the parallel computations of attention is called a head.

$$MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [head_1, \cdots, head_h]\mathbf{W}^O \tag{2}$$

where $head_i = Attention(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$, and every $\mathbf{W}$ represents a learnable parameter matrix. In addition, we also place the layer normalization between the residual blocks inside a transformer block as Formula (3) and (4) show.

$$Output = LayerNorm(Input + MultiHead(Input)) \tag{3}$$

$$Output = LayerNorm(Output + FFN(Output)) \tag{4}$$

where FFN represents the fully connected feed forward layer.

In a nutshell, we use the Transformer blocks as an encoder that can analyse the assembly code pair and generates a series of hidden states that capture the semantics and global context of the inputs. The subsequent linear layers finalize the prediction with outputting a similarity score. When training the model, we utilize Adam optimizer [16] with a sparse categorical cross-entropy loss function to improve the learning rate.

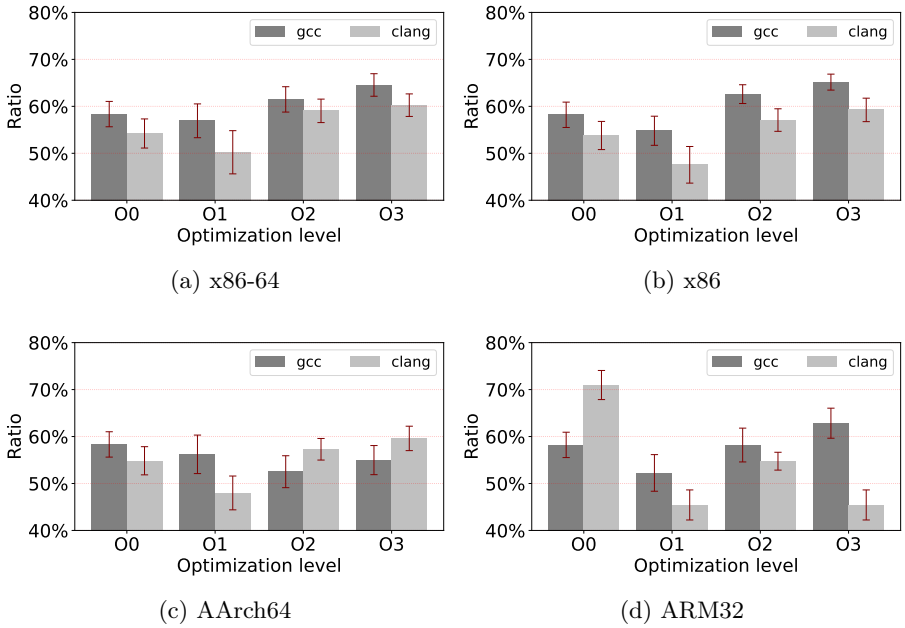
It is necessary to emphasize that the focus of this paper is on the construction of the dataset. Consequently, the the binary code similarity detector presented herein serves solely as a means to demonstrate BinSimDB's usage in reality. Based on this, we will showcase the advantage of our dataset over prior competitors in Section 4.3.

4 Evaluation

Employing the methodology introduced in Section 3, we build a fine-grained dataset BinSimDB as a benchmark for BCSA study. In this section, empirical studies are conducted to investigate the dataset properties. Also, the extensive evaluations showcase the strength of BinSimDB for the related research.

4.1 Dataset Properties and Composition

Diversity and Scalability. To generate binary samples, we collect source code of 30 binaries from 8 GNU software projects [12], i.e., `binutils`, `datamash`, `findutils`, `grep`, `gzip`, `macchanger`, `tar`, and `which`. They are all real programs



(a) x86-64 (b) x86

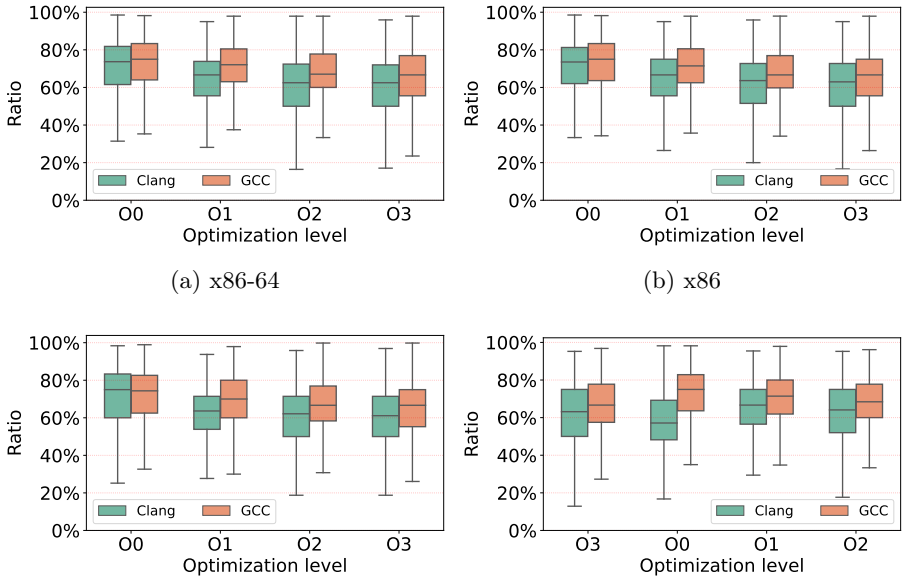
(c) AArch64 (d) ARM32

Fig. 7: The average ratio of the functions that need to be processed by BMerge to all functions, where the error bars indicate the 95% confidence intervals.

and widely deployed in the current software ecosystem. More importantly, because their source code is publicly accessible, GNU packages have became a very popular research resource for BCSA [2, 17, 31, 38]. Our comprehensive BCSA benchmark BinSimDB involves 980,251 functions across 32 distinct combinations of compilers, optimization levels, and target platforms. These functions will be used to further generate equivalent assembly code pairs. More specifically, we include binaries compiled for four different ISAs such as x86, x86-64, ARM32, and AArch64. Two representative compilers, i.e., GCC and Clang, are involved. We also consider four optimization levels from O0 to O3. On top of that, we develop automated scripts to compile the collected source code and disassemble the resulting binaries for all designated architectures and optimization levels. Therefore, other researchers can extend the existing dataset with little effort, or customize their dataset generation towards diverse application scenarios such as IoT applications analysis.

Fine Granularity. Unlike other existing work [38], we do not drop any assembly code obtained from the disassembler, so our dataset provides a good coverage. Not only that, owing to our proposed algorithms, we can generate semantically equivalent assembly code pairs at a finer granularity.

It is worth noting that not all functions need to be handled by the BMerge. If the source code snippets used to generate basic blocks are mutually exclusive, it is surely unnecessary to merge any such basic blocks. To quantitatively study



(a) x86-64 (b) x86

(c) AArch64 (d) ARM32

Fig. 8: The changes in the number of basic blocks after applying BMerge (Ratio $= \frac{\#\ of\ original\ basic\ blocks}{\#\ of\ resulting\ basic\ blocks}$).

the cases where a one-to-one mapping cannot be established between basic blocks and source code, we conducted program-level statistics on the proportion of the functions containing such basic blocks among all the functions. The results are shown in Figure 7. As is evident, the ratio of the functions that need to be processed by the BMerge to all functions is around 45% $\sim$ 65%, by average. On the ARM32 platform, when using `O3` as the optimization level, this ratio even reaches 70.9%. Therefore, it can be concluded that the phenomenon of lacking one-to-one mapping between basic blocks and source code is widespread in practice.

We also count the number of functions, on which we conduct BMerge, and show the statistics in Table 1. Based on this, we investigate the impact of applying the BMerge algorithm on the number of basic blocks, as Figure 8 shows. It can be seen that the distribution of changes in the number of basic blocks is diverse in terms of different architectures and optimization levels. However, the majority of functions are likely to see a reduction in the number of their basic blocks to approximately 80% or below of the original count.

For the same function coming from different architectures, optimization levels or compilers, we can efficiently extract equivalent basic blocks pairs by invoking the BPair algorithm. Table 2 and Table 3 show the resulting equivalent assembly code pairs targeting 64-bit and 32-bit platforms, respectively. It's worth

Table 1: The number of functions that need to be processed by BMerge. These functions are leveraged to investigate the impact of our proposed algorithm.

ISAs	**Compilers**	**Optimization levels**			
		O0	**O1**	**O2**	**O3**
x86	GCC	19,765	15,492	16,909	16,358
	Clang	18,565	17,571	13,529	14,067
x86-64	GCC	19,784	16,075	16,945	16,300
	Clang	19,061	19,167	14,513	14,942
ARM32	GCC	19,404	14,376	15,201	14,685
	Clang	25,349	16,348	12,525	13,620
AArch64	GCC	22,167	17,805	18,780	18,031
	Clang	21,720	20,196	15,218	15,906

Table 2: The number of equivalent pairs that are generated by the BPair algorithm under 64-bit architectures.

ISAs (Compilers)	ISAs (Compilers)	AArch64 (GCC)				AArch64 (Clang)			
	Opt levels	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**
x86-64 (GCC)	**O0**	35,691	34,364	32,632	31,303	39,760	44,917	31,073	30,331
	O1	34,702	42,953	36,911	39,557	33,844	33,384	36,420	35,538
	O2	34,148	40,884	39,750	37,134	32,658	32,068	35,188	34,239
	O3	32,440	37,926	37,397	41,367	30,909	30,279	35,667	34,841
x86-64 (Clang)	**O0**	35,961	32,430	30,559	29,253	38,640	44,452	30,488	29,799
	O1	44,603	32,212	30,187	28,706	45,092	45,550	31,263	30,478
	O2	30,895	35,327	33,538	33,801	30,973	31,461	42,820	41,626
	O3	30,032	34,238	32,646	33,008	30,084	30,563	41,553	41,763

Table 3: The number of equivalent pairs that are generated by the BPair algorithm under 32-bit architectures.

ISAs (Compilers)	ISAs (Compilers)	ARM32 (GCC)				ARM32 (Clang)			
	Opt levels	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**
x86 (GCC)	**O0**	38,833	32,740	31,099	32,639	40,515	43,235	29,914	28,684
	O1	33,052	38,410	38,124	35,414	31,744	30,601	32,998	31,484
	O2	31,474	35,606	36,512	34,443	29,950	28,413	31,526	30,217
	O3	30,310	33,167	33,845	37,675	28,733	27,306	31,860	30,585
x86 (Clang)	**O0**	37,095	31,599	31,686	30,161	37,773	44,006	30,506	29,446
	O1	44,589	31,719	30,935	29,336	44,758	44,195	30,686	29,438
	O2	30,995	34,734	34,584	35,203	31,029	30,613	42,247	40,385
	O3	30,227	33,785	33,581	34,291	30,281	29,697	40,825	40,469

noting that all resulting assembly pairs shown in Table 2 and Table 3 have been normalized and all duplicates have also been eliminated.

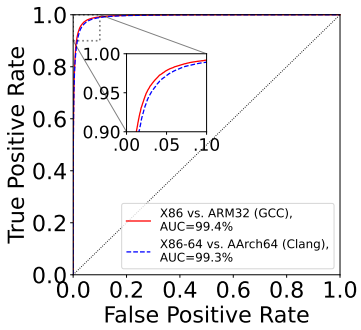
4.2 Application in Similarity Detection

The important application of BinSimDB is that it can be used to train machine learning models capable of detecting fine-grained similarities between binary code snippets. To illustrate this point, we attempt different data combinations from the proposed dataset, and train the Transformer-based detector introduced in 3.3. It should be clarified that, the basic blocks that have more than 100 instructions will be truncated when training the machine learning model, because we found only 0.45% of cases in our dataset fall into this category. Please note that what we do herein will also lay the groundwork for the subsequent experiments in Section 4.3 and Section 4.4.

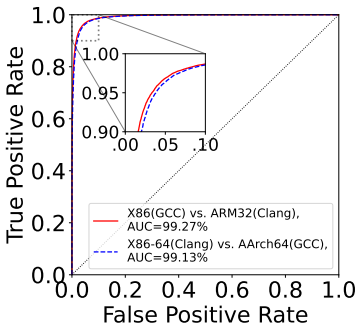
We first extract datasets towards 32-bit platforms, denoted as $\mathcal{D}$-`x86(GCC)/ARM32(GCC)`. In total, there are 750,000 basic block pairs in this dataset, where half are equivalent pairs while the other half are not. We randomly choose 80% instances as the training set and the remaining 20% as the testing set. Two basic blocks in a pair are across different optimization levels, and also different architectures, i.e., x86 and ARM32. Additionally, GCC is leveraged as a control variable, so all binaries in this dataset are built using the same compiler. Besides, we re-use the method proposed in [38] to produce nonequivalent basic block pairs. The evaluation result shows a well-trained model can finally achieve an AUC value as high as 99.4% over the testing set. The AUC (Area Under the Curve) value is a scale-invariant performance measurement for classification models. Specifically, it is used with Receiver Operating Characteristic (ROC) curves. A higher AUC value indicates a better measure of separability, that is the model effectively differentiates between the positive and negative classes.

Following the similar setting, we can obtain another dataset, which is towards 64-bit platforms, namely $\mathcal{D}$-`X86-64(Clang)/AArch64(Clang)`. Herein, the different aspect is two basic blocks in a pair are from x86-64 and AArch64 architectures, respectively. Plus, all binaries in the dataset are built using Clang. The evaluation result shows the model can achieve an AUC value as 99.3% over the testing set. The ROC curve is shown in Figure 9(a).

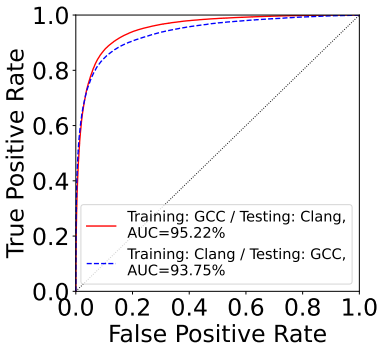
We further investigate the performance of models targeting cross-compiler challenge. To this end, we consider a dataset, $\mathcal{D}$-`x86(GCC)/ARM32(Clang)`. The size and distribution proportion of this dataset follow the same settings as aforementioned. However, two basic blocks in a pair not only come from different architectures, but also are obtained by different compilers. Namely, x86 and ARM32 binaries are built with GCC and Clang, respectively. The evaluation result shows the model can achieve a high AUC value as 99.27% over the testing set. Based on another dataset focusing on 64-bit platforms, $\mathcal{D}$-`x86-64(Clang)/AArch64(GCC)`, we also can observe an analogous performance, as Figure 9(b) shows. All in all, the performance of our models remains stable no matter which compilers or architectures are involved, proving that the models manifest better generalization ability.



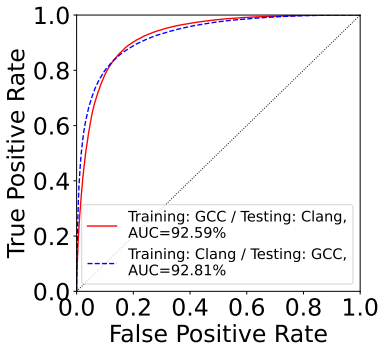
(a) Cross-ISAs & opt-levels



(b) Cross-ISAs, opt-levels & compilers

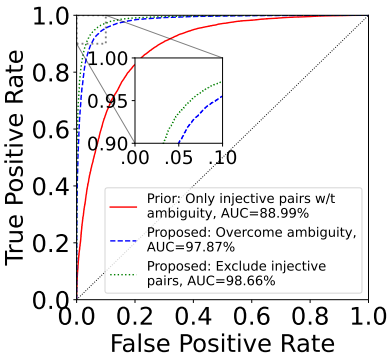


(c) x86 vs. ARM32

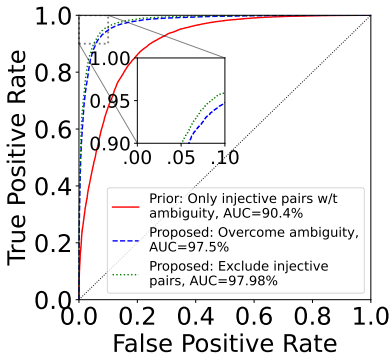


(d) x86-64 vs. AArch64

Fig. 9: Similarity detection among basic blocks based on BinSimDB.



(a) x86(GCC) vs. ARM32(GCC)



(b) x86-64(Clang) vs. AArch64(GCC)

Fig. 10: The advantage of BinSimDB over prior method [38].

Transferability. Different compilers will likely end up emitting noticeably distinct assembly code, though the obtained programs should still behave in exactly the same way. The discrepancy among compilers usually poses challenge in discovering semantic equivalence between binaries especially in the cross-architecture scenario. Therefore, we are interested in understanding the extent of transferability of a learning-based model can demonstrate when faced with previously unseen samples generated by a different compiler. For this purpose, we first construct two dis-joint datasets towards 32-bit platforms, $\mathcal{D}_{Train}$-`x86/ARM32 (GCC)` and $\mathcal{D}_{Test}$-`x86/ARM32(Clang)`. The size and distribution proportion of datasets follow the same settings as aforementioned. The change is that all binaries in the training set are compiled by GCC. In contrast, all binaries in the testing set are compiled by Clang. The evaluation result shows the model can achieve an AUC value of 95.22% over the testing set. If the binaries in the training set are compiled by Clang, while the binaries in the testing set are compiled by GCC. We can observe a very close AUC value of 93.75%, as Figure 9(c) shows.

We additionally conduct a complementary study to assess the extent to which this compiler-agnostic character can be maintained by a learning-based model towards binaries on 64-bit platforms. We first involve two dis-joint datasets, that is $\mathcal{D}_{Train}$-`x86-64/AARCH64(GCC)` and $\mathcal{D}_{Test}$-`x86-64/AARCH64(Clang)`. In other words, all binaries in the training set are compiled by GCC. However, the binaries in the testing set are compiled by Clang. The evaluation result shows the model can achieve an AUC value of 92.59% over the testing set. If we swap the compilers for the training and testing sets, the AUC value remains relatively stable, i.e., 92.81%, as Figure 9(d) shows.

4.3 Comparison Study

In the previous study such as [38], a pair of semantically equivalent basic blocks under distinct architectures across different optimization levels rely on simplified evidence to build a connection, that is the two basic blocks exclusively come from certain lines of source code and such source code are not used to generate any other basic blocks. If there is any ambiguity that cannot be straightforwardly handled, the basic blocks will be directly abandoned. Hence, a large number of semantically intense code gadgets may ultimately be excluded from the resulting dataset, such as the ternary operator expression shown in Figure 2.

To verify our insights, we extract such complicated cases to form a testing set towards 32-bit platforms, consisting of 43,000 basic block pairs. As Figure 10(a) shows, if we adopt the approach [38] to construct a training set, the classifier trained with this dataset can achieve an AUC value as 88.99% on the testing set. If we use the proposed BMerge and BPair to overcome the ambiguity, thus obtained training set can evidently improve the AUC value of the classifier to 97.87%. If the injective mapping between two cross-architecture basic blocks can be easily inferred, such a basic block pair is considered as a simple case. Even though we further exclude those simple cases from training set, the performance of trained model remains stable, with an AUC value of 98.66%. The size of all training sets is 172,000, so that the testing set takes up 20% of the overall data.

When considering 64-bit platforms, and cross-compiler scenarios, we can observe a similar result. As Figure 10(b) shows, by using BMerge and BPair to construct the training set, we can achieve an AUC value at 97.88% and 97.5% depending on whether the simple cases are removed or not. This is obviously higher than the comparative group with an AUC value of 90.4%. Therefore, we can conclude that the proposed dataset construction approach can greatly help to improve the performance of binary code similarity comparison.

4.4 Case Study

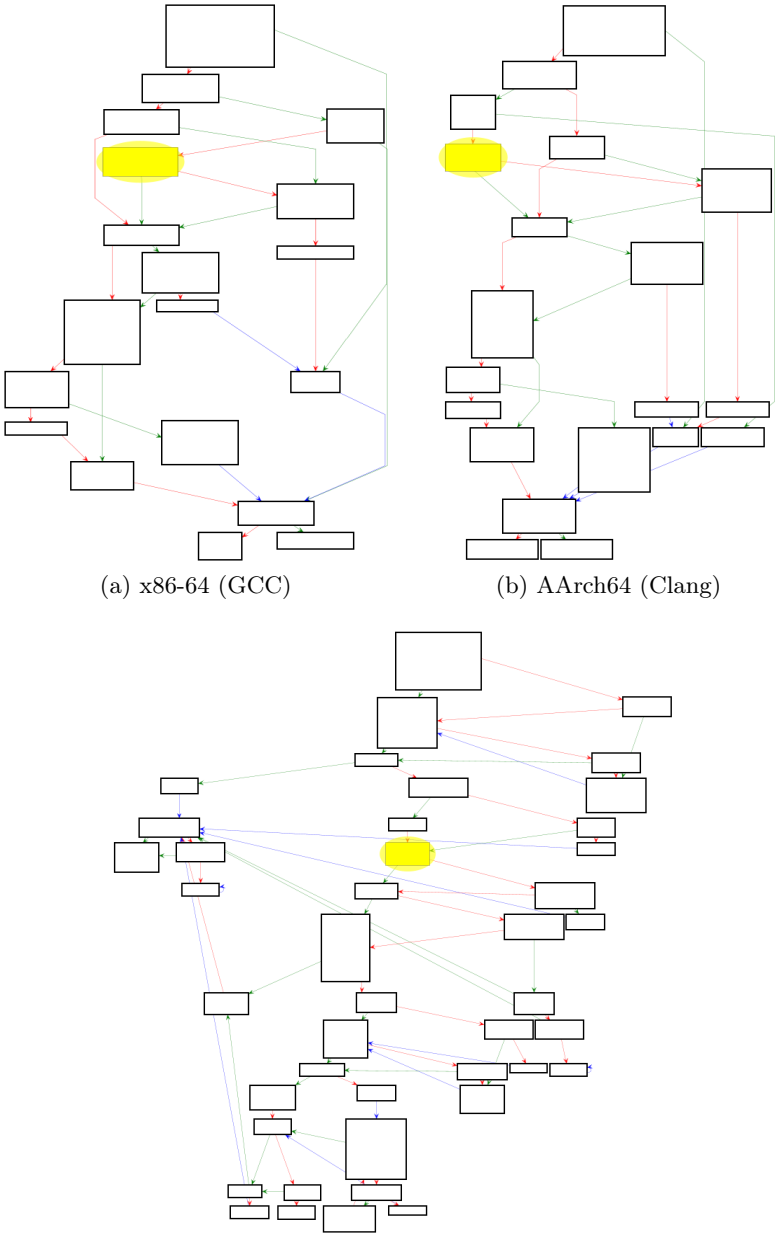
Again, the focus of this work is on the construction of the dataset, rather than developing new advanced BCSA techniques. To demonstrate the application of the *fine-grained* binary code similarity comparison in real-world scenarios, we conducted an experiment for the purpose of patch detection. CVE-2019-5482 is a heap buffer overflow vulnerability in the TFTP protocol handler of `libcurl`. The affected versions range from 7.19.4 to 7.65.3. Figure 11(a) shows the CFG of `tftp_connect` function from the upgraded version 7.66.0, where the patched basic block is highlighted in color. The binary is built in GCC towards an x86-64 platform, and the default optimization level is adopted. Locating the patch in a function with hundreds of basic blocks is a time-consuming task, but it is crucial for security engineers. The existing function-level BCSA approaches have difficulties in solving this kind of problems. Figure 11(b) shows the CFG of a patched function but from the version 7.67.0. We build the binary towards an AArch64 platform using Clang as the compiler. Our experiment shows that a detector introduced in Section 3.3, trained using the proposed fine-grained dataset, is able to quickly and accurately identify the affected basic block, namely the highlighted one. Moreover, we leverage the *bogus control flow insertion* [13] to generate an obfuscated binary, as Figure 11(c) shows. Still, our method can rapidly locate the corresponding basic block. This definitely could save a significant amount of time in security analysis. By contrast, we cannot find well-matched basic blocks in an earlier version such as 7.65.0 because it is unpatched. All the results are manually verified.

5 Related Work

5.1 Binary Similarity Analysis

The advances in applied deep learning, such as graph learning and natural language processing (NLP), have inspired many new methods for comparing the binary code similarity. Given a large body of research in the pertinent area, our literature review is not intended to be exhaustive.

Code Learning. Assembly code can be considered as a sequence of tokens, therefore numerous researchers shift their focus to the arsenal of NLP techniques when tackling the binary function similarity problem. For example, `BinDNN` [17], `Asm2Vec`, and `SAFE` [20], etc. Later, Yu et al. utilized the BERT model pre-trained



(a) x86-64 (GCC) (b) AArch64 (Clang)

(c) AArch64 (Clang) with obfuscation

Fig. 11: The CFGs of `tftp_connect` function in `libcurl`.

on four tasks to learn semantics of assembly code [37]. Similarly, `BinShot` [2] is also based on BERT model to detect binary code similarity. In addition, `jTrans` [31] proposed a binary function embedding method using the jump-aware Transformer-based model. However, all of these approaches detect similarity between functions, and cannot tackle the case when only partial code within a function is under consideration.

Graph Learning. There are a surge of works on binary code similarity analysis, which adopt existing graph based techniques. For example, `Structure2Vec` [8] is adapted by Xu et al. to learn CFG embeddings [34]. Besides, Yu et al. [37] use an adjacency matrix to represent a function CFG, then a CNN is further applied to generate embeddings. However, these methods suffer from a failure to capture the rich contextual information in assembly code (e.g., intra- or inter-procedural control flows). To address this problem, `DeepBinDiff` [10] builds inter-procedural CFG (ICFG) based on the call graph and CFGs to provide program-wide contextual information. In particular, Text-associated DeepWalk algorithm (TADW) [35] is used to learn vector representation for each vertex in a graph. Furthermore, Kim et al. [15] leverage the graph convolutional networks based graph alignment technique [33] to learn contextual information. However, this method relies on partial cross-platform alignment information as a priori.

5.2 Dataset Construction

Unlike computer vision and natural language which have a large body of well-labelled data, high-quality datasets are a kind of precious or even scare resource in cybersecurity. Many researchers consider the unavailability of well maintained data is a common barrier in the area of cybersecurity [28, 32, 39]. Therefore, we have seen a few recent works focusing on dataset construction, for example, system provenance dataset `ProvSec` [25], and source code patch dataset `PatchDB` [32]. However, the datasets targeting binary analysis are still a remaining issue which needs to be addressed by the security community. While some groups open-sourced their datasets, the contamination or degradation of data quality has been observed. For example, previous work [3] studied the dataset [4], which is used for the function boundaries detection in binary code, and "*found many functions to be duplicated across the training and testing sets, thus artificially increasing their F1-score*" [24]. In addition, the datasets for binary code similarity research are very rare. After investigating 43 papers in this area, Kim et al. [14] found "*only two of them opened their entire dataset*", which leads to be often infeasible for reproducing the previous results. In our previous research, we also encountered similar issues. For example, `CrossMal` [26] is a dataset consisting of cross-architecture function pairs based on IoT binaries. The download link provided in the paper had became inaccessible when we wrote this article.

A few new datasets for the binary similarity analysis task have been proposed in recent years [14,19,40]. However, these datasets cannot support a fine-grained study such as the similarity comparison at a basic block level. By contrast, Zuo et al. [38] propose an approach, which can automatically generate equivalent basic

block pairs. But, their method extracts binary code directly from the backend of compilers, which still has a certain gap from practical application because the binary code obtained from decompilation is not exactly identical to the binary code generated by the compiler.

5.3 Usage Cases of Binary Similarity

Binary similarity has been used for multiple usage cases. Vulnerability detection and patch detection is one of well-known application cases. We presented a case study to detect a vulnerability and a patch in our paper in the evaluation section. Another popular application is the detection and correlation of malicious software which are found in various fields across Enterprise [1, 36], IoT [7, 22], and Energy fields [6, 21]. To reach out to a large scope of targets, there are even types of malware that work across platforms (e.g., OS) and architectures. Binary similarity techniques will be a useful foundational technique to extend the applicability of the defense techniques.

6 Discussion and Future Work

The main purpose of BinSimDB is to facilitate BCSA research. However, because we have established a strong connection between assembly code and corresponding source information as the ground truth, the current dataset can be easily extended for diverse research purposes. For example, we can further build up a dataset containing semantically meaningful code gadgets, such as a loop structure. This dataset can be used to investigate the loop detection problem in binary analysis. We consider this as an interesting exploration direction.

Large Language Models (LLMs) have achieved remarkable success in various natural language processing tasks. As we have developed automated scripts that can continuously generate large volumes of high-quality data, this also paves the way for binary analysis research based on LLMs. We believe the intersection of LLMs and binary analysis has the potential to inspire further advancements in relevant areas, and accordingly plan to conduct more related study in the future.

7 Conclusion

This paper presents the construction of BinSimDB, a fine-grained dataset for research on binary code similarity analysis. To maintain the high quality of data, we extract ground truth from source code information. We propose two algorithms, BMerge and BPair, specifically designed to address the challenges of matching semantically equivalent binary code snippet pairs across different architectures and optimization levels. The proposed algorithms can preserve a good coverage and provide fine granularity to the greatest degree. Furthermore, we conduct comprehensive experiments to investigate the properties of the constructed dataset and demonstrate the strength of our proposed algorithms. The evaluation results show that BinSimDB is promising to facilitate the binary code similarity analysis.

Acknowledgement

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This article describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government. This work was supported through contract CR-100043-23-51577 with the U.S. Department of Energy.

References

1. Acar, A., Lu, L., Uluagac, A.S., Kirda, E.: An analysis of malware trends in enterprise networks. In: Lin, Z., Papamanthou, C., Polychronakis, M. (eds.) Information Security. pp. 360–380. Springer International Publishing, Cham (2019)
2. Ahn, S., Ahn, S., Koo, H., Paek, Y.: Practical binary code similarity detection with BERT-based transferable similarity learning. In: The 38th Annual Computer Security Applications Conference (ACSAC). pp. 361–374 (2022)
3. Andriesse, D., Slowinska, A., Bos, H.: Compiler-agnostic function detection in binaries. In: IEEE European Symposium on Security and Privacy (2017)
4. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: ByteWeight: Learning to recognize functions in binary code. In: The 23rd USENIX Security Symposium (2014)
5. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
6. Carter, C., Onunkwo, I., Cordeiro, P., Johnson, J.: Cyber security assessment of distributed energy resources. In: The 44th Photovoltaic Specialist Conference (PVSC). pp. 2135–2140 (2017)
7. Cozzi, E., Vervier, P.A., Dell'Amico, M., Shen, Y., Bilge, L., Balzarotti, D.: The tangled genealogy of IoT malware. In: The 36th Annual Computer Security Applications Conference (ACSAC). pp. 1–16 (2020)
8. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. In: International conference on machine learning (ICML). pp. 2702–2711 (2016)
9. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: IEEE Symposium on Security and Privacy. pp. 472–489 (2019)
10. Duan, Y., Li, X., Wang, J., Yin, H.: Deepbindiff: Learning program-wide code representations for binary diffing. In: Network and Distributed System Security Symposium (NDSS) (2020)
11. Gao, H., Cheng, S., Xue, Y., Zhang, W.: A lightweight framework for function name reassignment based on large-scale stripped binaries. In: The 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 607–619 (2021)
12. GNU Project: GNU packages. `https://www.gnu.org/software/software.en.html` (2023), accessed: 2023-03-01

13. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm–software protection for the masses. In: IEEE/ACM 1st international workshop on software protection. pp. 3–9 (2015)
14. Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Transactions on Software Engineering (2022)
15. Kim, G., Hong, S., Franz, M., Song, D.: Improving cross-platform binary analysis using representation learning via graph alignment. In: The 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (2022)
16. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: International Conference on Learning Representations (ICLR) (2015)
17. Lageman, N., Kilmer, E.D., Walls, R.J., McDaniel, P.D.: BinDNN: Resilient function matching using deep learning. In: The 12th International Conference on Security and Privacy in Communication Networks (SecureComm) (2016)
18. Li, X., Qu, Y., Yin, H.: Palmtree: Learning an assembly language model for instruction embedding. In: The ACM SIGSAC Conference on Computer and Communications Security. pp. 3236–3251 (2021)
19. Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D.: How machine learning is solving the binary function similarity problem. In: The 31st USENIX Security Symposium. pp. 2099–2116 (2022)
20. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: SAFE: Self-attentive function embeddings for binary similarity. In: The 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2019)
21. Mounet, F.B.: All your solar panels are belong to me. `https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Fred-Bret-Mounet-All-Your-Solar-Panels-Are-Belong-To-Me.pdf` (2024), accessed: 2024-06-22
22. Ngo, Q.D., Nguyen, H.T., Le, V.H., Nguyen, D.H.: A survey of IoT malware and detection methods based on static features. ICT Express **6**(4), 280–286 (2020)
23. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: IEEE symposium on security and privacy. pp. 833–851 (2021)
24. Qiao, R., Sekar, R.: Function interface analysis: A principled approach for function recognition in cots binaries. In: The 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2017)
25. Shrestha, M., Kim, Y., Oh, J., Rhee, J., Choe, Y.R., Zuo, F., Park, M., Qian, G.: ProvSec: Cybersecurity system provenance analysis benchmark dataset. In: The 21st IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (2023)
26. Song, Q., Zhang, Y., Wang, B., Chen, Y.: Inter-BIN: Interaction-based cross-architecture IoT binary similarity comparison. IEEE Internet of Things Journal **9**(20), 20018–20033 (2022)
27. The LLVM Project: LLVM online documentation. `https://llvm.org/doxygen/classllvm_1_1MachineBasicBlock.html` (2023), accessed: 2023-03-01
28. Tian, Y., Lawall, J., Lo, D.: Identifying Linux bug fixing patches. In: The 34th international conference on software engineering (ICSE) (2012)
29. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)

30. Votipka, D., Punzalan, M.N., Rabin, S.M., Tausczik, Y., Mazurek, M.L.: An investigation of online reverse engineering community discussions in the context of ghidra. In: IEEE European Symposium on Security and Privacy (2021)
31. Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C.: jtrans: jump-aware transformer for binary code similarity detection. In: The 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (2022)
32. Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S.: PatchDB: A large-scale security patch dataset. In: The 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2021)
33. Wang, Z., Lv, Q., Lan, X., Zhang, Y.: Cross-lingual knowledge graph alignment via graph convolutional networks. In: The Conference on Empirical Methods in Natural Language Processing (EMNLP) (2018)
34. 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: ACM SIGSAC conference on computer and communications security. pp. 363–376 (2017)
35. Yang, C., Liu, Z., Zhao, D., Sun, M., Chang, E.Y.: Network representation learning with rich text information. In: The International Joint Conference on Artificial Intelligence (IJCAI) (2015)
36. Yen, T.F., Heorhiadi, V., Oprea, A., Reiter, M.K., Juels, A.: An epidemiological study of malware encounters in a large enterprise. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 1117–1130 (2014)
37. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S.: Order matters: Semantic-aware neural networks for binary code similarity detection. In: The AAAI Conference on Artificial Intelligence (2020)
38. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. In: Network and Distributed System Security Symposium (NDSS) (2019)
39. Zuo, F., Rhee, J.: Vulnerability discovery based on source code patch commit mining: a systematic literature review. International Journal of Information Security **23**(2), 1513–1526 (2024)
40. Zuo, F., Rhee, J., Kim, Y., Oh, J., Qian, G.: A comprehensive dataset towards hands-on experience enhancement in a research-involved cybersecurity program. In: The 24th Annual Conference on Information Technology Education (SIGITE). pp. 118–124 (2023)