BinAligner: Aligning Binary Code for Cross-Compilation Environment Diffing

Yiran Zhu[1], Tong Tang[1], Jie Wan[1], Ziqi Yang*[1,2], Zhenguang Liu*[1,2], and Lorenzo Cavallaro[3]

[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University
[2]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
[3]University College London
{zhuyiran, tong.tang, wanjie, yangziqi, liuzhenguang}@zju.edu.cn, l.cavallaro@ucl.ac.uk

Abstract—**Binary diffing aims to align portions of control flow graphs corresponding to the same source code snippets between two binaries for software security analyses, such as vulnerability and plagiarism detection tasks. Previous works have limited effectiveness and inflexible support for cross-compilation environment scenarios. The main reason is that they perform matching based on the similarity comparison of basic blocks. In our work, we propose a novel diffing approach BinAligner to alleviate the above limitations at the binary level. To reduce the likelihood of false and missed matches corresponding to the same source code snippets, we present conditional relaxation strategies to find candidate subgraph pairs. To support a more flexible binary diffing in cross-compilation environment scenarios, we use instruction-independent basic block features for subgraph embedding generation. We implement BinAligner and conduct experiments across four cross-compilation environment scenarios (i.e., cross-version, cross-compiler, cross-optimization level, and cross-architecture) to evaluate its effectiveness and support ability for different scenarios. Experimental results show that BinAligner significantly outperforms the state-of-the-art methods in most scenarios. Especially in the cross-architecture scenario and multiple combinations of cross-compilation environment scenarios, BinAligner exhibits F1-scores that are on average 65% higher than the baselines. Two case studies using real-world vulnerabilities and patches further demonstrate the utility of BinAligner.**

I. Introduction

Binary code differential analysis, i.e., binary diffing [1]–[4], aims to align code portions corresponding to the same source code snippets between two binaries. It can play a vital role in a series of security scenarios caused by code cloning [5], [6], such as code plagiarism detection [7], [8], vulnerability and patch analysis [9], [10], and bug replication analysis [11]. Another technique called binary code similarity detection [12]–[26] is widely used in these security scenarios. In contrast to binary code similarity detection which learns feature representations to evaluate the similarity between two binaries, binary diffing focuses on finding correspondences to capture the differences between them. Therefore, the goal of binary diffing cannot be directly achieved using binary code similarity detection methods, which do not output correspondences. In this paper, we focus on aligning portions of the Control Flow Graph (CFG) that correspond to the same source code snippets between two given binary functions in cross-compilation environment scenarios.

* Corresponding authors: Ziqi Yang (yangziqi@zju.edu.cn) and Zhenguang Liu (liuzhenguang@zju.edu.cn)

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230649
www.ndss-symposium.org

The main challenge of binary diffing in cross-compilation environment scenarios is that basic blocks corresponding to the same source code may not always align precisely. This is because different compilation strategies in varying compilation environments may cause changes in divisions of basic blocks corresponding to the same source code. For example, a piece of source code may be split into multiple basic blocks in one compilation environment, while multiple basic blocks may be merged into one in another compilation environment (such as a higher optimization level).

Existing binary diffing methods, both traditional-based and Machine Learning (ML)-based, do not address this challenge. These methods rely on basic block matching, which involves computing the similarity of two basic blocks. Traditional methods usually perform basic block matching using instruction syntax and rely on fixed rules, which can easily be disrupted by changes in the compilation environment. These methods can be mainly divided into feature-based [1], [27], [28], symbolic execution-based [2], [29], graph isomorphism-based (i.e., matching nodes to find the largest common subgraph) [2], [30], and graph decomposition-based (i.e., decompose the graph into data-flow dependency chains within the scope of a basic block as the basic comparable unit of similarity) [31]–[33]. ML-based approaches [3], [4] achieve more accurate binary diffing than traditional methods due to stronger basic block feature learning capabilities. InnerEye [3] learns basic block embeddings by automatically capturing the semantics and dependencies of instructions with Long Short-Term Memory (LSTM) [34]. Then, it decomposes the CFG into multiple paths and matches basic blocks to find the longest common subpath. DeepBinDiff [4] generates basic block embeddings by considering instruction semantic information with Text-associated DeepWalk (TADW) algorithm [35]. It explores the neighbors of already-matched basic block pairs to match more basic blocks.

However, ML-based approaches still encounter two limitations. The first limitation is the restricted diffing effectiveness caused by the matching of only nodes. In particular, the source code is compiled into a graph (CFG) connected by nodes (basic blocks). The information of the same source code snippets is mapped to subgraphs of CFG and is unevenly dispersed along the edges to the nodes within the subgraph. This implies that while the cumulative quantity of information in two subgraphs remains constant, the information distributions of the nodes in the subgraphs may vary. Thus, calculating similarities of only node embeddings causes information loss. The second limitation is that these approaches cannot perform flexible cross-compilation environment diffing. InnerEye implements binary diffing across two architectures (i.e., x86 and arm). Each architecture pair requires re-training a complex neural network (with about 3.7 million parameters). For each binary pair, a corresponding neural network needs to be selected. DeepBinDiff does not support cross-architecture scenarios and only implements x86 diffing. This is because they heavily rely on architecture-dependent instruction information to learn node embeddings.

Performing diffing at a higher-level language (i.e., pseudocode) can mitigate the second limitation. The state-of-the-art pseudocode diffing approach SigmaDiff [36] constructs Intermediate Representation (IR) level Interprocedural Program Dependency Graphs (IPDGs). It matches nodes in IPDGs using a graph-matching model called Deep Graph Matching Consensus (DGMC) [37]. A node in the IPDG represents an IR statement, which is decompiled from binary code and is a smaller similarity calculation unit than the basic block. The decompiler strives to recover the semantic information of binary code through inference, which means it does not guarantee the consistency of IPDGs between different binaries of the same source code. Thus, SigmaDiff still encounters information loss due to performing node-level matching (i.e., the first limitation).

In this paper, we present a novel ML-based diffing approach BINALIGNER that aims to alleviate the above two limitations at the binary level. To improve the accuracy of aligning portions of CFG corresponding to the same source code snippets, we propose graph-level matching (i.e., finding two subgraphs and computing a similarity score of their graph embeddings) to replace node-level matching (i.e., finding two nodes and computing a similarity score of their node embeddings). To support a more flexible cross-compilation environment diffing on the binary language, we utilize basic block features independent of the instruction set as node embeddings for generating graph embeddings.

BINALIGNER can be divided into three steps. We repeat the first two steps alternately to obtain all potential candidate subgraph pairs and decide which candidate subgraph pairs are aligned in the third step. Specifically, given two CFGs, we divide the nodes into anchor nodes and chain nodes. An anchor node can be a branch node, an entry node, or an exit node (i.e., a node with special topological features). A chain node is an intermediate node that connects two anchor nodes, with the possibility of multiple chain nodes existing between them. We first enumerate nodes not belonging to matched subgraphs to identify an initial subgraph pair. The initial subgraph pair is composed of an anchor node pair (i.e., two nodes with the same in-degree and the same out-degree). Starting with the initial subgraph pair, we expand the subgraphs to obtain a candidate subgraph pair by iteratively incorporating neighboring nodes (including anchor nodes and chain nodes) through our proposed conditional relaxation strategies. Finally, we calculate similarities of candidate subgraph pairs by comparing their graph embeddings generated by cross-compilation environment node embeddings and an ML-based model.

We use graph-level information to identify the initial subgraph pair, i.e., the first node pair in the candidate subgraph pair. To improve matching accuracy, a node that serves as the initial subgraph is selected among the anchor nodes. Chain nodes are excluded during this step to reduce the probability of semantic irrelevance. We match initial subgraph pairs in which at most one anchor node belongs to matched candidate subgraphs to obtain more distinct initial subgraph pairs.

To expand an initial subgraph pair into a candidate subgraph pair, we propose two conditional relaxation strategies that iteratively incorporate neighboring nodes. The first strategy requires identifying new anchor node pairs along the predecessor and successor directions of the currently identified node pair, respectively. Chain nodes may exist between a new node pair and the current node pair to connect them. In each iteration, the new node pair and these potential chain nodes are incorporated into the subgraph pair. This strategy aims to absorb redundant nodes (e.g., empty instructions insertion, alignment padding) and adapt to chain deformations of the CFG (e.g., loop unrolling, jump instruction replacement). Then, we perform the second strategy on the subgraph pair expanded by the first strategy. If an anchor node is directly connected to any node in the current subgraph or indirectly connected through chain nodes, the anchor node and the potential chain nodes will be incorporated into the subgraph. This strategy aims to alleviate the mismatch caused by changes related to branches (e.g., basic block reordering, predication).

To measure the similarity of candidate subgraph pairs across cross-compilation environments, we use the instruction set independent basic block statistical attributes [22], [23] as node embeddings. The Siamese architecture [38] based on the graph embedding network Structure2vec [39] is then employed as a binary classifier to generate and compare graph embeddings.

We implement a prototype of BINALIGNER and conduct extensive experiments to evaluate its effectiveness and support for cross-compilation environment scenarios. Furthermore, we categorize the function pairs based on varying degrees of similarity. The effectiveness and support for various scenarios of BINALIGNER are also evaluated on function pairs with different similarity degrees. The experimental results show that BINALIGNER performs optimally in most scenarios, especially in the cross-architecture scenario and combination of multiple scenarios. For instance, in the cross-architecture scenario, BinAligner's F1-scores are more than 46.7% higher than the baselines. In the combination of cross-version and cross-optimization level scenarios based on function pairs with low similarity degrees, the F1-score of BinAligner is more than 3 times that of the baselines. Additionally, we demonstrate the strongest resilience to compilation environment evolution compared to the baselines. We further validate the utility of our approach via real-world vulnerability/patch case studies.

We summarize our contributions as follows:

- To our knowledge, we are the first to achieve graph-level matching for cross-compilation environment binary diffing, while previous works only perform node-level matching.
- In our proposed approach BinAligner[1], we reduce the probability of false and missed matches by designing conditional relaxation strategies. In addition, we support more flexible cross-compilation environment diffing at the binary level by using instruction set independent node embeddings to generate subgraph embeddings.
- Experimental results underscore the significant impact of BinAligner in the binary diffing task. It exhibits superior effectiveness and supports more complex cross-compilation environment scenarios than state-of-the-art methods. Furthermore, it shows greater resilience to compilation environment evolution. Case studies validate BinAligner's real-world vulnerability/patch identification utility.

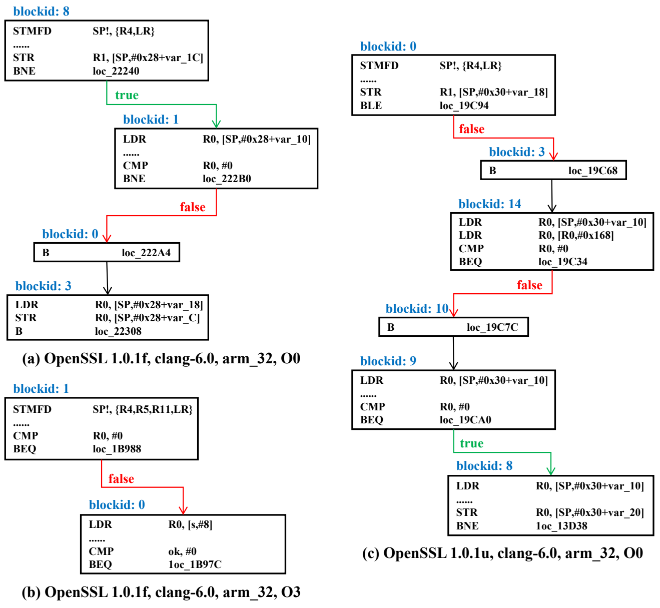
II. Problem Statement

In this section, we first formally define how to deal with the binary diffing problem using graph-level matching in our scope, then exhibit the motivation for graph-level matching through specific examples, and finally introduce state-of-the-art diffing techniques in detail.

A. Problem Definition

Given a binary function pair (f, f'), we denote their CFGs as $G = \langle \mathbb{V}, \mathbb{E} \rangle$ and $G' = \langle \mathbb{V}', \mathbb{E}' \rangle$, where $\mathbb{V}$ and $\mathbb{V}'$ are the sets of vertices, $\mathbb{E}$ and $\mathbb{E}'$ are the sets of edges. A subgraph of G is denoted as $G^s = \langle \mathbb{V}^s, \mathbb{E}^s \rangle$, where $\mathbb{V}^s \subseteq \mathbb{V}$ and $\mathbb{E}^s \subseteq \mathbb{E}$. The graph-level matching method is to first identify a set of candidate subgraph pairs (denoted as $\mathbb{P}_c^g = \{(G_i^s, G_i^{s\prime}) | 1 \leq i \leq m\}$) and then decide a set of aligned subgraph pairs (denoted as $\mathbb{P}_a^g$) from candidate subgraph pairs, where m is the number of candidate subgraph pairs and $\mathbb{P}_a^g \subseteq \mathbb{P}_c^g$. $\forall (G_j^s, G_j^{s\prime}) \in \mathbb{P}_a^g$, G_j^s and $G_j^{s\prime}$ correspond to the same source code snippet (i.e., same source code lines obtained by text matching).

Note that since we calculate similarities based on subgraph embeddings of CFG using static features of binary code, obfuscated code (e.g., packing and encryption) is not considered in this paper.



```
STMFD          SP!, {R4,LR}
......
STR            R1, [SP,#0x28+var_1C]
BNE            loc_22240
```

```
LDR            R0, [SP,#0x28+var_10]
......
CMP            R0, #0
BNE            loc_222B0
```

```
B              loc_222A4
```

```
LDR            R0, [SP,#0x28+var_18]
STR            R0, [SP,#0x28+var_C]
B              loc_22308
```

(a) OpenSSL 1.0.1f, clang-6.0, arm_32, O0

```
STMFD          SP!, {R4,R5,R11,LR}
......
CMP            R0, #0
BEQ            loc_1B988
```

```
LDR            R0, [s,#8]
......
CMP            ok, #0
BEQ            1oc_1B97C
```

(b) OpenSSL 1.0.1f, clang-6.0, arm_32, O3

```
STMFD          SP!, {R4,LR}
......
STR            R1, [SP,#0x30+var_18]
BLE            loc_19C94
```

```
B              loc_19C68
```

```
LDR            R0, [SP,#0x30+var_10]
LDR            R0, [R0,#0x168]
CMP            R0, #0
BEQ            loc_19C34
```

```
B              loc_19C7C
```

```
LDR            R0, [SP,#0x30+var_10]
......
CMP            R0, #0
BEQ            loc_19CA0
```

```
LDR            R0, [SP,#0x30+var_10]
......
STR            R0, [SP,#0x30+var_20]
BNE            1oc_13D38
```

(c) OpenSSL 1.0.1u, clang-6.0, arm_32, O0

Fig. 1: Partial CFGs of Function **ssl3_check_finished** under different compilation environments and OpenSSL versions.

B. Motivation for Graph-level Matching

In this section, we explain our motivation for performing graph-level matching instead of node-level matching. The compilation of the same source code can produce different CFGs based on different compilers, compiler optimization levels, and architectures. For example, a line of source code may correspond to two basic blocks under one compilation environment but may be optimized into one basic block under another compilation environment. Fig. 1a and 1b respectively illustrate partial CFGs of a function **ssl3_check_finished** in OpenSSL [40] in the same version (1.0.1f) but with different compilation environments (clang-6.0, arm_32, O0 vs clang-6.0, arm_32, O3). According to the mapping between the source code line number and the program address obtained from debug symbol information, the basic blocks with id 1 and 3 in Fig. 1a align the basic block with id 0 in Fig. 1b. Furthermore, the compilation of different source codes (such as differences caused by version upgrades), can yield different CFGs. Within the same compilation environment, alterations to the copied function or selective content-copying may result in changes to the CFG of the entire function. Fig. 1c displays a partial CFG of **ssl3_check_finished** under version 1.0.1u with the same compilation environment as Fig. 1a. The basic block with id 8 in Fig. 1a aligns the basic blocks with id 0 and 9 in Fig. 1c. These cases mean that node-level matching can lead to missed and false matches. In contrast, graph-level matching increases the probability of a correct and comprehensive alignment. Thus, we perform graph-level matching to improve the probability of aligning binary code portions corresponding to the same source code snippets.

[1]https://github.com/zyyrrr/BinAligner

C. *State-of-the-art Techniques*

The state-of-the-art techniques [3], [4], [36] achieve diffing based on the binary language and the pseudocode language.

Binary Diffing. InnerEye [3] calculates the similarity of two basic blocks from x86 and arm architectures respectively referring to machine translation. It regards instructions as words and proposes a cross-lingual basic block embedding model based on LSTM [34]. Then, it decomposes the CFG into a set of linearly independent paths using the Depth First Search (DFS) algorithm. Finally, it matches basic blocks to find the longest similar path between a query path and a target path by adopting Breadth First Search (BFS) combined with the Longest Common Subsequence (LCS) dynamic programming algorithm. DeepBinDiff [4] compares two basic blocks both from x86 binaries. It first generates token (i.e., opcode or operand) embeddings using a token embedding model derived from the Word2Vec technique [41] and then generates basic block embeddings by iteratively optimizing token embeddings with the TADW algorithm [35]. Finally, it matches basic blocks within the k-hop neighbors of already matched ones.

Pseudocode Diffing. SigmaDiff [36] searches for node pairs with high similarity in IR-level IPDGs. It first extracts node features using a lightweight symbolic analysis. Then, it generates node embeddings and matches nodes leveraging the ML-based DGMC model [37]. A node (i.e., an IR statement) at the IR level is a smaller unit than a node (i.e., a basic block) at the binary level.

Overall, they all perform node-level matching, so the effectiveness of diffing is limited by missed and false matches. In addition, binary diffing methods are difficult to be flexible across compilation environments due to instruction reliance.

III. Approach

A. BinAligner *Overview*

The overview of BinAligner is shown in Fig. 2. Given a pair of binary functions, we use the mainstream disassembly tool IDA Pro [42] to generate the target CFG pair. BinAligner consists of three steps. We repeat the first two steps alternately to find potential candidate subgraph pairs and decide aligned subgraph pairs in the third step. In the first step, we identify an initial subgraph pair by enumerating nodes that do not belong to matched subgraphs. To reduce false matches, the initial subgraph pair comprises two nodes with special topological features, sharing the same in-degree and the same out-degree. In the second step, we obtain a candidate subgraph pair by expanding the initial subgraph pair. To reduce missed matches, we propose conditional relaxation strategies that iteratively incorporate neighboring nodes. To make the matched subgraphs larger to reduce omissions, we allow partial overlap between different candidate subgraph pairs. Meanwhile, candidate subgraph pairs remain distinguishable from each other because of the differences in their initial subgraph pairs. In the third step, we decide aligned subgraph pairs by calculating the similarity between the graph embeddings of candidate subgraph pairs. To generate graph embeddings that are independent of instructions, we utilize basic block statistical attributes [22], [23] as node embeddings. The Siamese network [38] based on Structure2vec [39] is used to learn and compare graph embeddings.

B. *Initial Subgraph Pair Identification*

Given a CFG pair, we identify initial subgraph pairs by enumerating nodes in a random order after the preprocessing of eliminating loops. To reduce the probability of false subgraph matches, we select anchor nodes to construct initial subgraphs.

1) Anchor/Chain Node Definition: To effectively exploit graph-level information, we divide nodes in CFG into two categories: anchor nodes and chain nodes. Branch nodes, entry nodes, and exit nodes are classified as anchor nodes. Intermediate nodes between anchor nodes are defined as chain nodes, which are used to connect anchor nodes. Compared with chain nodes, anchor nodes have special topological features. In particular, the topological features of all chain nodes are consistent, i.e., the in-degree and out-degree are both equal to 1. It can be formalized as:

$$(\mathbf{In}(n) = 1) \wedge (\mathbf{Out}(n) = 1) \tag{1}$$

where n represents a node in the CFG, $\mathbf{In}(\cdot)$ and $\mathbf{Out}(\cdot)$ are functions for counting a given node's in-degree and out-degree, respectively. In contrast, anchor nodes may exhibit diverse topological features. However, they all share the property that at least one of their in-degree or out-degree is not equal to 1 (i.e., = 0 or $>$1), which can be formalized as:

$$(\mathbf{In}(n) \neq 1) \vee (\mathbf{Out}(n) \neq 1) \tag{2}$$

Fig. 3b exemplifies anchor/chain node distinction: solid boxes denote anchor nodes (red: branch nodes, purple: entry nodes, orange: exit nodes); dashed boxes indicate chain nodes.

2) Loop Elimination Preprocessing: To ensure the presence of anchor nodes, we preprocess the two given CFGs to eliminate their loops consisting only of chain nodes. An example of a CFG missing anchor nodes is that it has only one node and an edge pointing to itself. After removing this edge, the chain node becomes an anchor node. To avoid non-unique preprocessing results that may cause uncertainty in subsequent steps, loop elimination preprocessing is fixed rather than heuristic. Specifically, we traverse and record all chain nodes. Starting from an unrecorded chain node, we recursively record its successors. If the recursion eventually reaches the starting node, it indicates the presence of a loop formed solely by chain nodes. In this situation, we remove the edge between the currently reached recorded node and its predecessor node.

3) Initial Subgraph Pair Matching: We identify the initial subgraph pair comprising only one node pair (i.e., $(G^s = \{n_0\}, G^{s\prime} = \{n_0'\})$) among all anchor nodes. This is because anchor nodes with the same topological features are more likely to be semantically related than chain nodes. To identify more distinct initial subgraph pairs, we allow the node pair in which at most one node belongs to already matched candidate subgraphs in the same CFG. Specifically, we specify three conditions to identify the initial subgraph pair. Condition 1

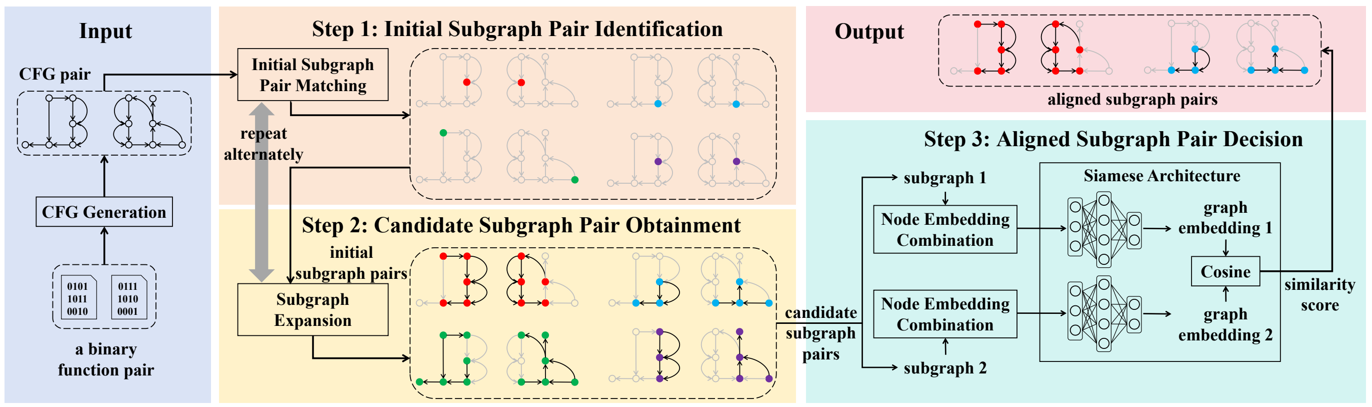


Fig. 2: Overview of BinAligner.

requires that the node pair has the same in-degree and the same out-degree. This condition is formalized as follows:

$$\mathbf{C1}: \quad (\mathbf{In}(n_0) = \mathbf{In}(n_0')) \wedge (\mathbf{Out}(n_0) = \mathbf{Out}(n_0')) \tag{3}$$

Condition 2 restricts that the node pair are both anchor nodes, which can be formalized as:

$$\begin{aligned}\mathbf{C2}: \quad &((\mathbf{In}(n_0) \neq 1) \vee (\mathbf{Out}(n_0) \neq 1)) \\ &\wedge((\mathbf{In}(n_0') \neq 1) \vee (\mathbf{Out}(n_0') \neq 1))\end{aligned} \tag{4}$$

Condition 3 demands that at least one node in the node pair does not belong to any matched candidate subgraphs. Suppose that we are currently identifying the i-th pair of initial subgraphs, Condition 3 can be formalized as:

$$\mathbf{C3}: \quad (n_0 \notin \bigcup_{j=1}^{i-1} G_j^s) \vee (n_0' \notin \bigcup_{j=1}^{i-1} G_j^{s\prime}) \tag{5}$$

where $\{(G_j^s, G_j^{s\prime})|1 \leq j \leq i-1\}$ represents the candidate subgraph pairs identified and expanded previously in the target CFG pair. Each initial subgraph pair satisfies the above three conditions, i.e., $\mathbf{C1}(n_0, n_0') \wedge \mathbf{C2}(n_0, n_0') \wedge \mathbf{C3}(n_0, n_0')$.

C. Candidate Subgraph Pair Obtainment

We expand an initial subgraph pair to obtain a candidate subgraph pair by iteratively incorporating neighboring nodes. To reduce the probability of missed subgraph matches, we introduce two conditional relaxation strategies.

1) Challenges of Tackling Missed Matches: Node-level matching omissions stem from CFG alterations (basic blocks cannot be aligned precisely) after the compilation environment changes. Table I presents several typical CFG changes that arise from variations in the compilation environment. These changes can be divided into chain (block splitting/merging) and branch (linearization/complication) changes. The challenge brought by the chain change is the linear dispersion/concentration of basic block information. To address this challenge, we propose the first relaxation strategy (denoted as $St.^{1st}$) to achieve linear information consolidation via subgraph matching. The challenge brought by the branch change lies in the alteration of basic blocks' topological positions (the branches to which they belong). Since the branch change occurs near unchanged branches, we propose the second strategy (denoted as $St.^{2nd}$) to perform anchor-based neighborhood search. This allows basic block information to be identified when the new position is still nearby. However, optimizations that scatter basic blocks across branches may cause some irrecoverable losses. We present the robustness of different conditional relaxation strategies to these CFG changes in Table I. $St.^{1st}$ adapts to changes such as loop optimization, dead code elimination, etc. $St.^{2nd}$ is robust to changes such as basic block reorganization and predication. However, BinAligner may miss some correct matches (e.g., exception handling code optimization). Note that our goal is to effectively reduce omission/error probabilities rather than completely avoid them.

2) The First Conditional Relaxation Strategy $St.^{1st}$: This strategy iteratively searches for new anchor node pairs based on the current anchor node pair. We compare whether the next two anchor nodes in the predecessor or successor direction of the current anchor node pair have the same in-degree and the same out-degree. If a new anchor node pair can be identified, the iteration in the current direction continues. Otherwise, the iteration in the current direction stops. Chain nodes may exist between the new anchor node pair and the current one. The relaxation of $St.^{1st}$ is reflected in incorporating the new anchor node pair and these potential chain nodes into the subgraph pair. This relaxation can improve the robustness against changes in the compilation environment that may cause the appearance of redundant nodes (e.g., empty instructions insertion, alignment padding) and chain deformations (e.g., loop unrolling, jump instruction replacement) in the CFG.

Specifically, we use five conditions to expand the initial subgraph pair. We denote the set of p pairs of anchor nodes identified in the t-th round of iteration as $\mathbb{P}_t^n = \{(n_i^t, n_i^{\prime t})|1 \leq i \leq p\}$. Thus, the set of q pairs of anchor nodes in the $t+1$-th round is $\mathbb{P}_{t+1}^n = \{(n_i^{t+1}, n_i^{\prime t+1})|1 \leq i \leq q\}$, which can be identified through Condition 1 and 2 as well as three new conditions. Condition 4 requires that a new pair of n_i^{t+1} and $n_i^{\prime t+1}$ does not belong to the subgraph pair currently being

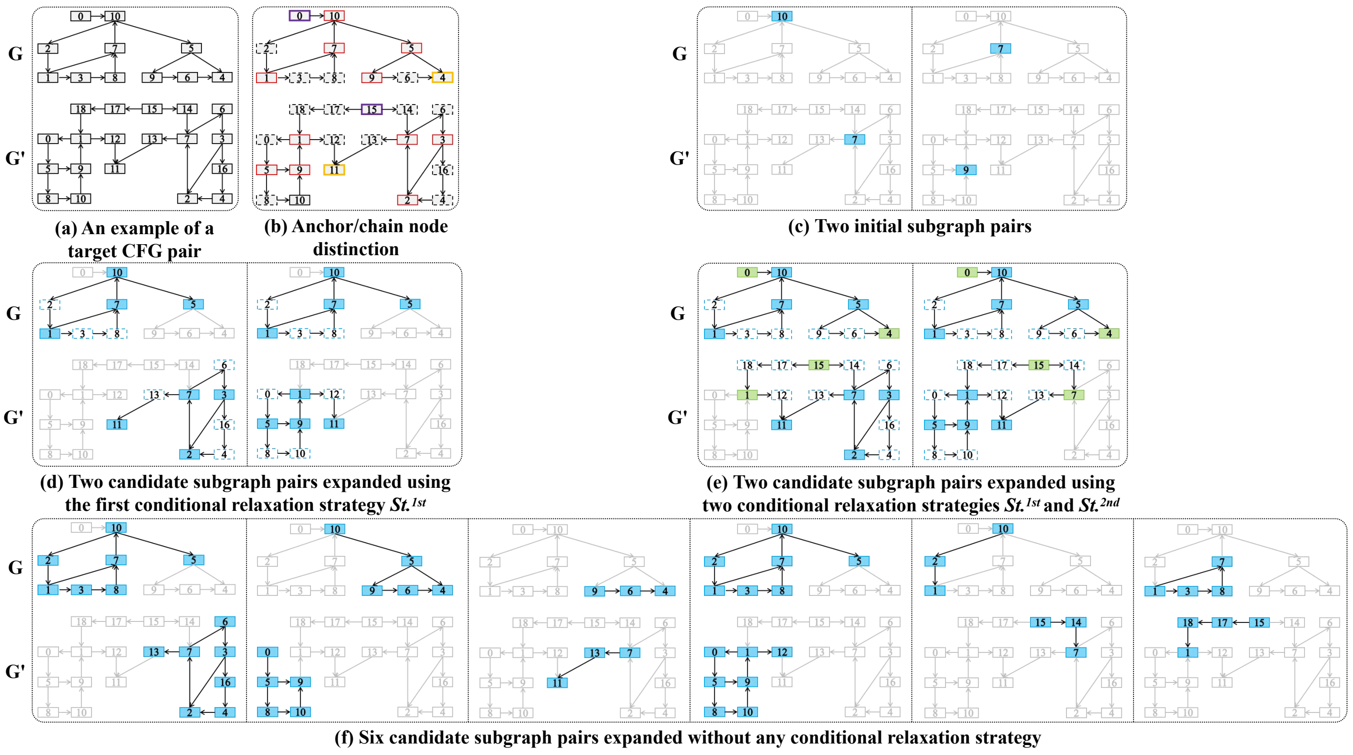

G
G'
(a) An example of a target CFG pair
(b) Anchor/chain node distinction
(c) Two initial subgraph pairs
(d) Two candidate subgraph pairs expanded using the first conditional relaxation strategy St.1st
(e) Two candidate subgraph pairs expanded using two conditional relaxation strategies St.1st and St.2nd
(f) Six candidate subgraph pairs expanded without any conditional relaxation strategy

Fig. 3: Candidate subgraph pairs for a target CFG pair.

TABLE I: Robustness of Conditional Relaxation Strategies for Different CFG Changes ($NoSt.$ / $St.^{1st}$ / $St.^{1st+2nd}$ denotes that BINALIGNER obtains candidate subgraph pairs without any conditional relaxation strategy / with the first conditional relaxation strategy / with two conditional relaxation strategies, ✓ means a certain ability, $\times$ means no ability)

Change	Partial CFG 1	Partial CFG 2	$NoSt.$	$St.^{1st}$	$St.^{1st+2nd}$
Loop Optimization	$A \rightarrow B \rightarrow C$	$A \rightarrow B \rightarrow B \rightarrow B \rightarrow C$	$\times$	✓	✓
Dead Code Elimination	$A \rightarrow B \rightarrow C$	$A \rightarrow C$	$\times$	✓	✓
Conditional Branch Simplification	$A \rightarrow true \rightarrow B$ $\searrow false \rightarrow C$	$A \rightarrow case1 \rightarrow B$ $\searrow case2 \rightarrow C$ $\searrow case3 \nearrow$	$\times$	✓	✓
Function Inlining	$A \rightarrow B \rightarrow C$	$A \rightarrow D \rightarrow E \rightarrow C$	$\times$	✓	✓
Tail Call Optimization	$A \rightarrow B \rightarrow C \rightarrow D$ $E \swarrow$	$A \rightarrow B \rightarrow F$ $E \swarrow \quad \nwarrow \vert$	$\times$	✓	✓
Basic Block Reorganization	$\cdots \rightarrow A \rightarrow B \rightarrow C \rightarrow \cdots$ $\searrow D \searrow \cdots$	$\cdots \rightarrow B \rightarrow A \rightarrow C \rightarrow \cdots$ $\searrow D \quad \searrow \cdots$	$\times$	$\times$	✓
Predication	$A \rightarrow B \rightarrow \cdots$ $\cdots \swarrow \searrow \cdots$	$A \rightarrow B \rightarrow \cdots$ $\searrow \cdots$	$\times$	$\times$	✓
Exception Handling Code Optimization	$(try)A \rightarrow (catch)B \rightarrow C$	$(try)A \rightarrow (catch1)B \rightarrow C$ $\searrow (catch2)D \rightarrow E$	$\times$	$\times$	$\times$

expanded. Suppose that the j-th pair of candidate subgraphs $(G_j^s, G_j^{s\prime})$ is being expanded, Condition 4 is formalized as:

$$\textbf{C4}: \quad (n_i^{t+1} \notin G_j^s) \wedge (n_i^{\prime t+1} \notin G_j^{s\prime}) \tag{6}$$

Condition 5 demands that $\exists(n_e^t, n_e^{\prime t}) \in \mathbb{P}_t^n$, n_i^{t+1} and $n_i^{\prime t+1}$ are both predecessor nodes or both successor nodes of n_e^t and $n_e^{\prime t}$, which can be formalized as:

$$\begin{aligned}\textbf{C5}: \quad &((n_i^{t+1} \in \textbf{Pred}(n_e^t)) \wedge (n_i^{\prime t+1} \in \textbf{Pred}(n_e^{\prime t}))) \\ &\vee((n_i^{t+1} \in \textbf{Succ}(n_e^t)) \wedge (n_i^{\prime t+1} \in \textbf{Succ}(n_e^{\prime t})))\end{aligned} \tag{7}$$

where $\textbf{Pred}(\cdot)$ and $\textbf{Succ}(\cdot)$ are functions for recording a given node's predecessor node set and successor node set, respectively. Condition 6 permits that n_i^{t+1} ($n_i^{\prime t+1}$) and n_e^t ($n_e^{\prime t}$) can be connected through chain nodes. Let l denote the number of chain nodes, where $l \geq 0$. The relationship between

n_i^{t+1} and n_e^t is formalized as follows, and the relationship between $n_i^{'t+1}$ and $n_e^{'t}$ is the same.

$$\mathbf{C6}: \quad (n_i^{t+1} \rightarrow n_1^c \rightarrow n_2^c \rightarrow ... \rightarrow n_l^c \rightarrow n_e^t) \\ \vee (n_e^t \rightarrow n_1^c \rightarrow n_2^c \rightarrow ... \rightarrow n_l^c \rightarrow n_i^{t+1}) \tag{8}$$

where n^c represents a chain node, and $\rightarrow$ represents an edge connecting two nodes. Each anchor node pair in the $t+1$-th round satisfies $\mathbf{C1}(n_i^{t+1}, n_i^{'t+1}) \wedge \mathbf{C2}(n_i^{t+1}, n_i^{'t+1}) \wedge \mathbf{C4}(n_i^{t+1}, n_i^{'t+1}) \wedge \mathbf{C5}(n_i^{t+1}, n_i^{'t+1}) \wedge \mathbf{C6}(n_i^{t+1}) \wedge \mathbf{C6}(n_i^{'t+1})$.

3) The Second Conditional Relaxation Strategy $St.^{2nd}$: This strategy searches for neighboring anchor nodes of the subgraph pair expanded using $St.^{1st}$. For each subgraph in the subgraph pair, a new anchor node is searched in the predecessor or successor direction along each anchor node that serves the boundary of the subgraph. The new anchor node is directly connected to the subgraph or indirectly connected through chain nodes. The relaxation of $St.^{2nd}$ is reflected in incorporating these new anchor nodes and potential chain nodes into the subgraph pair. This relaxation can improve the robustness against changes in the compilation environment that may cause CFG deformations related to branches (e.g., basic block reordering, predication).

Specifically, we use two conditions to expand the subgraph pair. We denote the set of r new anchor nodes searched in G using $St.^{2nd}$ as $\mathbb{N} = \{n_i^{T+1} | 1 \leq i \leq r\}$, where T represents the number of iterations in $St.^{1st}$. $\forall n_i^{T+1} \in \mathbb{N}$, n_i^{T+1} satisfies Condition 7 and 8, which are formalized as follows. The same applies to $n_i^{'T+1}$ searched in G'.

$$\mathbf{C7}: \ ((\mathbf{In}(n_i^{T+1}) \neq 1) \vee (\mathbf{Out}(n_i^{T+1}) \neq 1)) \wedge (n_i^{T+1} \notin G_i^s) \tag{9}$$

$$\mathbf{C8}: \ \exists n_x \in G_i^s, (n_i^{T+1} \rightarrow n_1^c \rightarrow n_2^c \rightarrow ... \rightarrow n_l^c \rightarrow n_x) \\ \vee (n_x \rightarrow n_1^c \rightarrow n_2^c \rightarrow ... \rightarrow n_l^c \rightarrow n_i^{T+1}) \tag{10}$$

Each neighboring anchor node searched in G and G' satisfies $\mathbf{C7}(n_i^{T+1}) \wedge \mathbf{C8}(n_i^{T+1})$ and $\mathbf{C7}(n_i^{'T+1}) \wedge \mathbf{C8}(n_i^{'T+1})$.

Fig. 3 illustrates BinAligner's initial subgraph pair identification and candidate subgraph pair obtainment: (a) a target CFG pair (G, G'); (c) initial subgraph pairs; (d) subgraph pairs expanded with $St.^{1st}$; (e) candidate subgraph pairs expanded with two conditional relaxation strategies. Here, the anchor node pairs identified using $St.^{1st}$ are marked with blue boxes, the anchor nodes searched using $St.^{2nd}$ are marked with green boxes, and the chain nodes incorporated into the subgraphs are represented by blue dashed boxes. In addition, Fig. 3f shows six candidate subgraph pairs expanded without any conditional relaxation strategy. The subgraph pairs are expanded iteratively by exploring neighboring node pairs (whether anchor node or chain node) with the same in-degree and the same out-degree. In this example, we can see the strategies bias toward larger subgraphs, reducing candidate count while enhancing graph-level information corresponding to the same source code snippets.

D. Aligned Subgraph Pair Decision

We decide the aligned subgraph pairs by calculating the graph embedding similarity of the two subgraphs in each candidate subgraph pair. To generate architecture-agnostic graph embeddings, we use instruction-independent basic block features as node embeddings.

We first generate the node embedding by adopting seven basic block attributes used in Gemini [22] and Genius [23]. These attributes include six block-level attributes (i.e., String/Numeric Constants, No. of Transfer/Calls/Arithmetic Instructions, No. of Instructions) and an inter-block attribute (i.e., No. of offspring), all quantified when the disassembler extracts the CFG. These statistical attributes focus on the overall behavior of a basic block and are independent of the instruction set of a specific compilation environment. Therefore, the graph embeddings learned based on these node embeddings can support similarity comparisons across compilation environments. For a subgraph, we combine the node embeddings in the order of node numbers. The disassembler takes care of the node numbering. Then, we input node embedding combinations of a candidate subgraph pair into an ML-based model. The model is the Siamese architecture [43], where the graph embedding network Structure2vec [39] serves as the generator of the subgraph embedding. It output a similarity score calculated by the cosine function. Finally, we obtain the aligned subgraph pairs according to a similarity threshold θ (see Section IV-D4 for details on threshold selection).

E. BinAligner *Process*

The binary diffing process of BinAligner is detailed in Algorithm 1. Given a target CFG pair, BinAligner performs preprocessing to eliminate loops (Ln. 1). Formal binary diffing consists of three steps. The first step is to identify an initial subgraph pair. Then, BinAligner expands initial subgraphs to obtain a candidate subgraph pair. The alternate execution of these two steps stops when the first step cannot generate a new initial subgraph pair (Ln. 2-25). In the third step, BinAligner calculates the similarity scores of candidate subgraph pairs (Ln. 26-35) to decide aligned subgraph pairs (Ln. 36).

IV. Evaluation

In this section, we first describe the experimental setup. For cross-compilation environment scenarios, we evaluate the effectiveness, support ability for different scenarios, and compilation environment evolution resilience of BinAligner. Next, we evaluate BinAligner's efficiency. Furthermore, we perform ablation studies to analyze the influence of different components and hyperparameters on BinAligner. We also show BinAligner's security applications by conducting case studies.

A. Experimental Setup

The experiments are conducted on a desktop computer with Ubuntu 22.04LTS operating system, 3.0 GHz Intel® Xeon® Gold 6248R CPU, 251GB memory, and two NVIDIA GeForce RTX 3090 GPUs. Additionally, we use Kunpeng&Ascend server platform for partial data processing and intermediate data storage, especially for large executable files. We use IDA Pro 7.5 [42] as the disassembly tool and implement BinAligner based on Python 3.7.

Algorithm 1 BinAligner Process

```
Input: $(G, G')$: the target CFG pair; $\mathbf{F}$: aligned subgraph pair decider; $\theta$: similarity threshold of graph embeddings
Output: the set of aligned subgraph pairs $\mathbb{P}_a^g$
 1: $G = \mathbf{EliminateLoop}(G)$, $G' = \mathbf{EliminateLoop}(G')$
 2: $\mathbb{G} = \mathbf{GetNodes}(G)$, $\mathbb{G}' = \mathbf{GetNodes}(G')$
 3: $\mathbb{P}_c^g = \{\}$
 4: while $\mathbb{G} \neq \emptyset$ do
 5:     $n_0 = \mathbb{G}.\mathbf{pop}()$
 6:     $\mathbb{G}'^* = \mathbf{GetNodes}(G')$
 7:     for $i = 0; i < ||\mathbb{G}'^*||; i++$ do
 8:         $n_0' = \mathbb{G}'^*[i]$
 9:         if $\mathbf{C1}(n_0, n_0') \wedge \mathbf{C2}(n_0, n_0') \wedge \mathbf{C3}(n_0, n_0')$ then
10:             $G^s = \{n_0\}, G^{s\prime} = \{n_0'\}$
11:             $(G^s, G^{s\prime}) = \mathbf{ExpandSubgraph}(G^s, G^{s\prime})$
12:             $\mathbb{P}_c^g.\mathbf{add}((G^s, G^{s\prime}))$
13:             $\mathbb{G} = \mathbb{G} - \mathbf{GetNodes}(G^s)$
14:             $\mathbb{G}' = \mathbb{G}' - \mathbf{GetNodes}(G^{s\prime})$
15:         end if
16:     end for
17: end while
18: while $\mathbb{G}' \neq \emptyset$ do
19:     $n_0' = \mathbb{G}'.\mathbf{pop}()$
20:     $\mathbb{G}^* = \mathbf{GetNodes}(G)$
21:     for $i = 0; i < ||\mathbb{G}^*||; i++$ do
22:         $n_0 = \mathbb{G}^*[i]$
23:         Same as Ln.9-15.
24:     end for
25: end while
26: $\mathbb{P}_a^g = \{\}$
27: for $i = 0; i < ||\mathbb{P}_c^g||; i++$ do
28:     $(G^s, G^{s\prime}) = \mathbb{P}_c^g[i]$
29:     $\vec{X} = \mathbf{GetNodeEmbeddings}(G^s)$
30:     $\vec{X'} = \mathbf{GetNodeEmbeddings}(G^{s\prime})$
31:     $s = \mathbf{F}(\vec{X}, \vec{X'})$
32:     if $s \geq \theta$ then
33:         $\mathbb{P}_a^g.\mathbf{add}((G^s, G^{s\prime}))$
34:     end if
35: end for
36: return $\mathbb{P}_a^g$
```

TABLE II: Number of Functions, Basic Blocks, and Control Edges extracted from Binary Files of GNU and OpenSSL Datasets

Library	Function	Basic Block	Control Edge
Coreutils	766,951	4,322,352	3,782,819
Diffutils	33,317	229,731	206,313
Findutils	30,667	250,983	232,433
OpenSSL	2,125,098	17,673,532	16,781,138

1) Datasets: We compile three libraries from the GNU project (called the GNU dataset) and OpenSSL [40] (called the OpenSSL dataset), which are widely used in practice and binary diffing research. IDA Pro extracts functions, basic blocks, and control edges from binary files of the GNU and OpenSSL datasets, with statistical breakdowns in Table II.

GNU Dataset. We compile Coreutils (5.93, 6.4, 7.6, 8.1, 8.25, 8.30, 9.1) [44], Diffutils (2.8, 3.1, 3.3, 3.4, 3.6) [45], and Findutils (4.41, 4.6) [46]. Compilation environments encompass GCC-5.4/8.2 and Clang-3.8 compilers, O0-O3 optimization levels, and x86-64/ARM-64 architectures. In total, there are 2,941 binary files in this dataset.

In the cross-version scenario, we use the optimization level of O1, the GCC-5.4 compiler, and the x86-64 architecture. Version comparisons are conducted for Coreutils ({5.93, 6.4, 7.6, 8.1, 8.25} vs 8.30), Diffutils ({2.8, 3.1, 3.3, 3.4} vs 3.6), and Findutils ({4.41} vs 4.6). In the cross-optimization level scenario, the versions are Coreutils 8.30, Diffutils 3.6, and Findutils 4.6. The compiler and architecture are the same as in the cross-version scenario. The optimization level comparisons conducted are {O0, O1, O2} vs O3. In the cross-compiler scenario, the optimization level is O0. The version and architecture are the same as the cross-optimization level scenario. The compiler comparison is GCC-5.4 vs Clang-3.8. In the cross-architecture scenario, the versions and compiler are the same as the cross-optimization level scenario, and the optimization level is O0. The architecture comparison is x86-64 vs ARM-64. To evaluate the compilation environment evolution resilience of BinAligner, we compile an updated version of Coreutils (Coreutils 9.1) in the cross-architecture scenario using a newer compiler (GCC-8.2) and a higher optimization level (O3) compared to the original compilation environment (Coreutils 8.30, GCC-5.4, O0).

We obtain function pairs with the same function name and then remove duplicate function pairs (follow the benchmark standard [47]) via source filenames and line numbers. The number of function pairs in the above five scenarios is 4,530, 4,272, 2,250, 2,377, and 935, respectively. Function pairs for the first four scenarios are randomly split 6:2:2 to train/test the ML-based aligned subgraph pair decision model and evaluate BinAligner. Function pairs of the fifth scenario are all used for the evolution resilience evaluation. To ensure DeepBinDiff compatibility, only function pairs in which two functions are from the same binary file are retained in the evaluation set. Note that we conservatively label subgraph pairs as aligned when their source code overlaps in at least one line.

OpenSSL Dataset. We compile two versions (1.0.1f and 1.0.1u) and obtain large executable files (i.e., *libssl.so* and *libcrypto.so*) using 2 compilers (GCC and Clang), of which GCC has 5 versions (4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0) and Clang has 2 versions (6.0, 7.0), 4 optimization levels (O0, O1, O2, O3), 3 architectures (x86, ARM, and MIPS) with 2-word sizes (32-bit and 64-bit). We acquire 2*2*(5+2)*4*3*2=672 binary files in total.

Functions are randomly sampled from 336 binary files of versions 1.0.1f and 1.0.1u, respectively, to form cross-version function pairs. In each function pair (f, f'), f comes from 1.0.1f and f' comes from 1.0.1u. Additionally, both f and f' come from either *libssl.so* or *libcrypto.so*. In this dataset, the comparison scenarios are denoted as X(V), X(V+C), X(V+O), and X(V+A). Specifically, function pairs in the X(V) scenario only cross versions, with the compilation environments remaining consistent. In the X(V+C/O/A) scenario, function pairs not only cross versions but also cross compilers/optimization levels/architectures, while the remaining compilation environments remain consistent. The compilation environment pairs for each scenario are 336, 2,016, 1,008, and 1,680, respectively. Take 2,016 as an example, which corresponds to X(V+C). We use 2 versions of OpenSSL and 7 compilers (different types or versions). The OpenSSL version pair is fixed to (1.0.1f, 1.0.1u), and the number of compiler pairs is 7*7-7=42. Therefore, 1*42*4(optimization levels)*3(architectures)*2(word sizes)*2(binary files)=2,016.

We divide function pairs into different similarity degree ranges according to the similarity rate R_s. R_s represents the proportion of subgraphs corresponding to the same snippets of source code within a target CFG pair. The calculation formula is detailed in Section IV-A4. We choose function pairs with similarity rates falling within four ranges: 10∼30%, 30∼50%, 50∼70%, and 70∼90%. Since function pairs are selected randomly, there are fewer function pairs with higher similarity rates. Let α represent the number of function pairs in the 70∼90% similarity degree range. To ensure an even distribution of samples across different similarity degree ranges, we randomly select α pairs of functions from the other three ranges. Thus, the numbers of function pairs in X(V), X(V+C), X(V+O), and X(V+A) are 12,086*4=48,344, 61,640*4=246,560, 32,466*4=129,864, and 61,283*4=245,132, respectively, totaling 669,900. We denote the set of these function pairs as D_f. Then, we sample 20,000, 40,000, 40,000, and 40,000 function pairs in the four scenarios, respectively. We denote the set of these function pairs as D_f^1. We sample one of the candidate subgraph pairs from each function pair in D_f^1, aiming to collect 20,000 pairs of aligned subgraphs and 20,000 pairs of unaligned subgraphs. These 40,000 subgraph pairs come from different function pairs and are divided into the training set and test set for the aligned subgraph pair decision model at a ratio of 8:2. Since none of the baselines conduct binary diffing on MIPS in their works, we ensure a fair comparison by sampling function pairs compiled on x86 and ARM from the remaining pairs in D_f to form a new dataset denoted as D_f^2. The number of function pairs in D_f^2 under the four scenarios is 2,000, 8,000, 8,000, and 8,000, respectively. Note that the training of baseline models does not use functions on MIPS [3], [4], [36], while BINALIGNER does.

2) Baseline Methods: We compare BINALIGNER (abbreviated as **BA**) with four baseline methods based on three state-of-the-art diffing techniques (described in Section II-C).

InnerEye (IE). We use the model provided by the authors and follow the setting of the basic block similarity threshold (i.e., 0.5). Additionally, we reproduce the LCS algorithm described in the paper.

DeepBinDiff (DBD). We use open-source code provided by the authors and follow the setting of the basic block similarity threshold (i.e., 0.6) and the number of hops (i.e., k=4).

DeepBinDiff-k-hop with InnerEye-embedding (DBD$^{\mathrm{IE}}$). Note that DeepBinDiff cannot be applied to the cross-architecture scenario and fails to process large binary files (i.e., *libssl.so* and *libcrypto.so*). Therefore, we apply DeepBinDiff's k-hop greedy matching algorithm on InnerEye's basic block embeddings with k=4 and the basic block embedding similarity threshold of 0.6. The algorithm extracts direct neighbors of virtual nodes in the Inter-procedural Control Flow Graph (ICFG) as initial matched node pairs. To adapt this algorithm for CFGs, we modify an initial node pair to a pair of nodes that exhibit maximal basic block embedding similarity (1.0).

SigmaDiff-DGMC with Gemini-embedding (SD$^{\mathrm{G}}$). To make SigmaDiff (the pseudocode diffing tool) suitable for binary diffing, we transform symbolic analysis node embeddings into Gemini's basic block embeddings and then use DGMC to match nodes. We utilize the code and hyperparameters of the DGMC model provided by the authors.

For convenience, we use their acronyms instead of full names in the following experimental analysis.

3) Ablation Study Settings: **Hyperparameters of BA's Algorithm and ML-based Model.** We employ randomized node enumeration ordering for initial subgraph pair identification, establishing reproducibility through a fixed random seed (0). Gemini's graph embedding network [22] is employed to generate candidate subgraph embeddings. The similarity threshold θ of the aligned subgraph pair decision model is 0.7. The node embedding adopts the best-performing combination of seven basic block attributes from the hyperparameter selection evaluation in Gemini [22]. In addition, training epochs of the model is 100 and the Adam [48] optimizer with a learning rate of 0.0001 is used to optimize the model parameters (following the default values of [22]). **Key Components of BA.** We evaluate BA with different conditional relaxation strategies and different subgraph embedding generators in the ablation study.

- BINALIGNER without any strategy (BA-0). In the second step of BA, we expand subgraph pairs to obtain candidate subgraph pairs without any conditional relaxation strategy. It means that the subgraph pairs are expanded iteratively by exploring neighboring node pairs (whether anchor node or chain node) that exhibit the same in-degree and the same out-degree.
- BINALIGNER with only the first strategy $St.^{1st}$ (BA-1). In the second step of BA, we expand subgraph pairs to obtain candidate subgraph pairs with only the first conditional relaxation strategy.
- BINALIGNER with GMN [49] (BA-GMN). In the third step of BA, we generate candidate subgraph embeddings using GMN. The hyperparameter settings of GMN are the same as those of Gemini's graph embedding network.

Same as baselines, we use their acronyms in the following experimental analysis.

4) Metrics: We follow and improve the metrics of DeepBinDiff [4] to be suitable for evaluating BINALIGNER. For a function pair (f, f'), we denote their basic block sets as U and U'. The ground truth sets of nodes in subgraphs corresponding to the same source code snippets are represented as T and T'. The nodes in aligned subgraph pairs (when using BA) and the matched nodes (when using the baseline methods) are indicated as M and M'. Then, the sets of correctly aligned nodes are denoted as $M_c = M \cap T$ ($M'_c = M' \cap T'$), the sets of incorrectly aligned nodes are denoted as M_i and M'_i, and the sets of unknown nodes are denoted as M_u

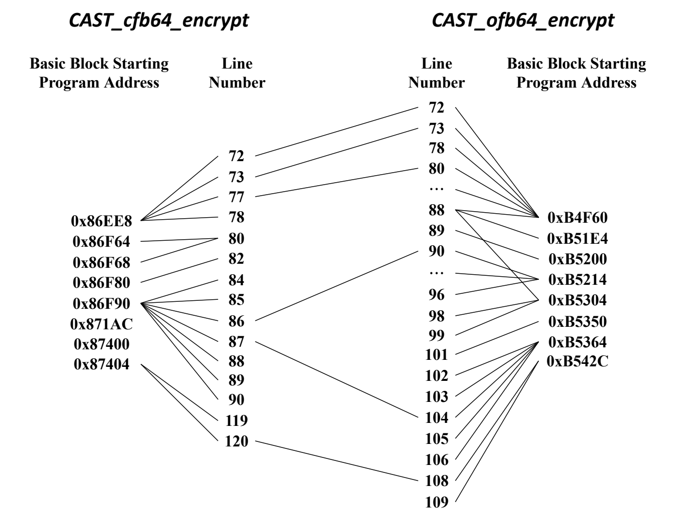


Fig. 4: The mapping relationship between basic block starting program addresses and source code line numbers.

and M'_u. We have no idea whether the unknown nodes are aligned correctly or incorrectly. This could happen due to the conservative method of gathering the same source codes via text matching. Thus, the relationship between M_c, M_i and M_u is $M_c + M_i + M_u = M$. We use Precision (P), Recall (R), and F1-score ($F1$) as metrics. p represents the proportion of correctly aligned nodes among all known aligned nodes.

$$p = \begin{cases} 0, \text{if } M = M_u \ or \ M' = M'_u \\ \frac{1}{2} \cdot (\frac{||M_c||}{||M - M_u||} + \frac{||M'_c||}{||M' - M'_u||}), \text{otherwise} \end{cases} \tag{11}$$

r means the ratio of correctly aligned nodes among nodes in subgraphs corresponding to the same source code snippets.

$$r = \frac{1}{2} \cdot (\frac{||M_c||}{||T||} + \frac{||M'_c||}{||T'||}) \tag{12}$$

We calculate the average value (i.e., P and R) of all ps and rs for a compilation environment scenario. Then, $F1$ is calculated by $\frac{2*P*R}{P+R}$. Furthermore, to measure the similarity degree of two functions, we define the similarity rate (R_s), which employs the ratio of nodes in subgraphs corresponding to the same source code snippets.

$$R_s = \frac{1}{2} \cdot (\frac{||T||}{||U||} + \frac{||T'||}{||U'||}) \tag{13}$$

5) Ground Truth Collection: For the binary diffing task, we rely on source code text matching and debugging symbol information to conservatively collect ground truth (i.e., the sets of nodes in subgraphs corresponding to the same source code snippets of two functions). Specifically, we extract two types of information from the debugging symbol information. (I) Function name, source file name, and source file path. These are employed to identify the same source code snippets between two functions. We use Python's *difflib* module [50] to perform text matching on the source code, thereby obtaining the ground truth of source code line numbers. This method ensures soundness, although the unmatched text may also be semantically identical. (II) The mapping between basic block starting program addresses and line numbers. Unlike [4], which deletes code statements that lead to multiple basic blocks, we retain all mapping relationships between program addresses and line numbers. Through the ground truth of the same source codes and the mapping relationship between program addresses and line numbers, we obtain the ground truth at the CFG-level.

Take two functions **CAST_cfb64_encrypt** and **CAST_ofb64_encrypt** as an example. The former is derived from *c_cfb64.c* in OpenSSL 1.0.1f, compiled with the options clang-6.0, arm_32, O0. The latter originates from *c_ofb64.c* in OpenSSL 1.0.1u, compiled using the settings clang-6.0, mips_64, O0. The mapping relationship between their basic block starting program addresses and line numbers is shown in Fig. 4. In **CAST_cfb64_encrypt**, Ln.82 $\leftrightarrow$ 0x86F80 is an one-line-to-one-address mapping, Ln.80 $\leftrightarrow$ (0x86F64, 0x86F68) and (Ln.119, Ln.120) $\leftrightarrow$ 0x87404 are multiple-lines-to-multiple-addresses mappings. Moreover, 0x871AC and 0x87400 have no corresponding line numbers. Through the mapping between line numbers, the alignment (0x86EE8, 0x86F90, 0x87404) $\leftrightarrow$ (0xB4F60, 0xB5214, 0xB5364, 0xB542C) of the two functions at the CFG-level can be obtained.

B. Effectiveness, Support Ability, and Evolution Resilience for Cross-compilation Environment Scenarios

We benchmark BA and baseline methods across four cross-compilation environment scenarios using the GNU dataset, evaluating the effectiveness, support ability for different scenarios, and compilation environment evolution resilience. Then, we evaluate the performance of these methods for function pairs spanning varying similarity degree ranges using the OpenSSL dataset.

1) Cross-version Diffing: In this experiment, we compare the performance of BA and four baselines in function pairs with different versions. As shown in Table III, BA outperforms the baselines in most version comparison cases, achieving an average $F1$ improvement of 39.5%, and up to 66.4%. This result shows that the diffing effectiveness of graph-level matching is better than that of node-level matching. When comparing BA with the suboptimal DBD$^{\text{IE}}$, we can see in Coreutils that the performance gap between the two methods becomes larger as the comparison versions become closer. When the comparison version is 5.93 vs 8.30, BA's $F1$ is slightly lower than that of DBD$^{\text{IE}}$ by 2.7%. This is because a larger version gap results in smaller snippets of the same source code, which means that the corresponding subgraph is also smaller. At the same time, the NN-based node embedding encompasses more instruction semantic information than the statistical attribute-based node embedding. Therefore, DBD$^{\text{IE}}$ can work when the version gap is large in the same architecture. However, when the version gap becomes smaller, $F1$s of graph-level matching gradually exceed those of node-level matching, and its advantages are more clearly demonstrated.

TABLE III: Cross-version Diffing Results

	Approach	Coreutils (* vs 8.30)					Diffutils (* vs 3.6)				Findutils
		5.93	6.4	7.6	8.1	8.25	2.8	3.1	3.3	3.4	4.41 vs 4.6
R	IE	0.209	0.237	0.285	0.268	0.25	0.299	0.28	0.239	0.294	0.209
	DBD	0.254	0.247	0.241	0.185	0.21	0.205	0.391	0.533	0.51	0.698
	DBD^{IE}	**0.773**	0.768	0.823	0.796	0.837	0.779	0.828	0.864	0.861	0.898
	SD^{G}	0.162	0.154	0.328	0.307	0.402	0.366	0.377	0.351	0.359	0.323
	BA	0.745	**0.782**	**0.881**	**0.864**	**0.947**	**0.834**	**0.961**	**0.904**	**0.972**	**0.967**
	BA-0	0.651	0.688	0.779	0.79	0.905	0.749	0.906	0.844	0.944	0.764
	BA-1	0.644	0.683	0.781	0.776	0.92	0.752	0.884	0.853	0.956	0.819
	BA-GMN	0.644	0.7	0.846	0.803	0.932	0.752	0.886	0.862	0.97	**0.967**
P	IE	0.6	0.63	0.672	0.656	0.669	0.564	0.645	0.656	0.67	0.697
	DBD	0.688	0.607	0.598	0.534	0.571	0.562	0.667	0.867	0.913	**1.0**
	DBD^{IE}	**0.968**	**0.963**	0.966	0.966	0.976	0.875	0.944	0.978	0.978	**1.0**
	SD^{G}	0.493	0.452	0.638	0.657	0.723	0.667	0.694	0.701	0.668	0.571
	BA	0.945	0.95	**0.976**	**0.974**	**0.996**	**0.906**	**1.0**	**0.989**	**1.0**	**1.0**
	BA-0	0.856	0.882	0.928	0.946	0.979	0.859	0.971	0.967	0.986	0.786
	BA-1	0.853	0.877	0.93	0.931	0.977	0.849	0.96	0.954	**1.0**	0.857
	BA-GMN	0.83	0.88	0.945	0.902	0.975	0.823	0.889	0.944	0.989	**1.0**
$F1$	IE	0.31	0.344	0.4	0.381	0.364	0.391	0.39	0.35	0.409	0.322
	DBD	0.371	0.351	0.344	0.275	0.307	0.3	0.493	0.66	0.654	0.822
	DBD^{IE}	**0.86**	0.855	0.889	0.873	0.901	0.824	0.882	0.917	0.916	0.946
	SD^{G}	0.244	0.23	0.434	0.418	0.517	0.473	0.489	0.468	0.467	0.413
	BA	0.833	**0.858**	**0.926**	**0.916**	**0.971**	**0.869**	**0.98**	**0.945**	**0.986**	**0.983**
	BA-0	0.74	0.773	0.847	0.861	0.941	0.8	0.937	0.901	0.965	0.775
	BA-1	0.734	0.768	0.849	0.846	0.948	0.798	0.92	0.901	0.978	0.838
	BA-GMN	0.725	0.78	0.893	0.85	0.953	0.786	0.887	0.901	0.979	**0.983**

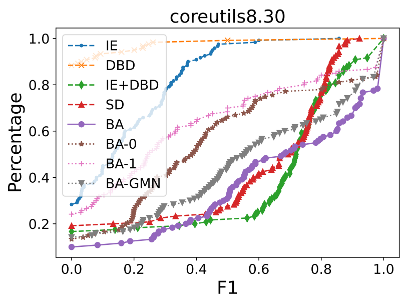
TABLE IV: Cross-compiler Diffing Results (GCC vs Clang)

	Approach	Coreutils 8.30	Diffutils 3.6	Findutils 4.6
R	IE	0.271	0.241	0.262
	DBD	0.04	0.045	0.053
	DBD^{IE}	**0.801**	**0.801**	0.808
	SD^{G}	0.177	0.161	0.242
	BA	0.797	**0.801**	**0.868**
	BA-0	0.655	0.707	0.731
	BA-1	0.684	0.694	0.748
	BA-GMN	0.773	0.752	0.781
P	IE	0.733	0.641	0.623
	DBD	0.172	0.197	0.177
	DBD^{IE}	**0.996**	**1.0**	**0.99**
	SD^{G}	0.541	0.553	0.604
	BA	0.991	0.992	**0.99**
	BA-0	0.974	0.951	0.945
	BA-1	0.949	0.93	0.938
	BA-GMN	0.963	0.942	0.902
$F1$	IE	0.396	0.35	0.369
	DBD	0.065	0.073	0.082
	DBD^{IE}	**0.888**	**0.89**	0.89
	SD^{G}	0.267	0.25	0.345
	BA	0.883	0.886	**0.925**
	BA-0	0.783	0.811	0.824
	BA-1	0.795	0.795	0.832
	BA-GMN	0.858	0.836	0.837

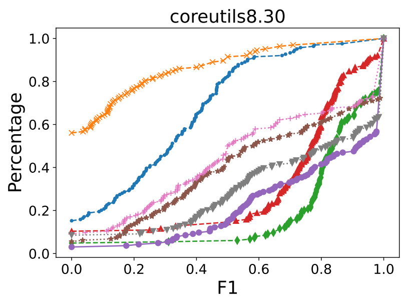
2) Cross-optimization-level Diffing: We then perform experiments to evaluate the effectiveness in the cross-optimization level scenario. Fig. 5 presents the Cumulative Distribution Function (CDF) figures of $F1$. The y-axis represents the proportion of $F1$ less than or equal to a certain value in all $F1$ values. We can observe that BA performs best in most cases, especially when $F1$ is in the range of 0.8 $\sim$ 1.0. This result shows the effectiveness of graph-level matching. When $F1$ is in the range of 0.4 $\sim$ 0.8, CDF curves of DBD^{IE} and SD^{G} are located below that of BA in some cases. This result illustrates that DBD^{IE} and SD^{G} usually correctly match nodes corresponding to a portion of the same source code snippets, but also have incorrect matches at the same time. In addition, it can be seen that the effectiveness of all diffing methods improves as the compared optimization levels become closer.

3) Cross-compiler Diffing: We also conduct cross-compiler diffing to evaluate BA. Table IV shows that $F1$ of BA in Findutils is 3.5% $\sim$ 84.3% higher than baselines while slightly lower (0.4% $\sim$ 0.5%) than DBD^{IE} in Coreutils and Diffutils. This is mainly because Ps of DBD^{IE} are slightly higher than BA. These results show that graph-level matching can improve the effectiveness, although the statistical attribute-based node embedding used to support cross-architecture may not perform as well as the NN-based node embedding under the same architecture. An interesting finding is that DBD performs poorly ($F1$ at most 8.2%). This is because DBD processes instructions based on GCC, which means that DBD cannot work across compilers beyond its inability to work across architectures. Additionally, SD^{G} underperforms relative to originally reported metrics in [36], exhibiting 13.4%/12.1% R and 3.3%/1.5% P reductions on identically compiled Diffutils 3.6/Findutils 4.6. This discrepancy stems from two transformations. (i) Transitioning from node-level to graph-level Ground Truth collection expands the basic block corpus, increasing recall's denominator (Eq. 12). (ii) Converting symbolic analysis node embeddings to basic block embeddings for binary diffing.

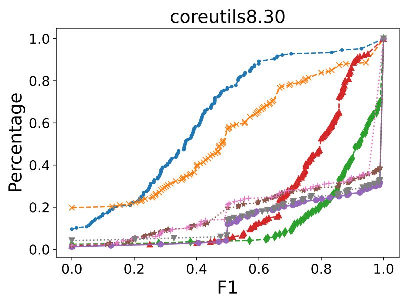
4) Cross-architecture Diffing: We compare BA with three baselines in the cross-architecture scenario, since DBD only supports x86 binaries. The results are shown in Table V. We can see that BA performs best. The highest R of these baselines is 90.9% and the highest P is 73.7%, while all metrics of BA are above 93% and most of them are close



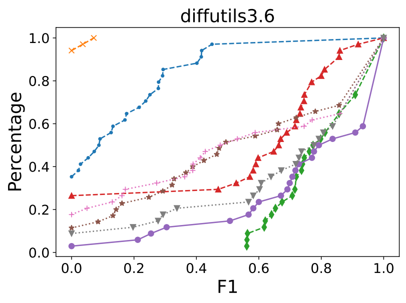
(a) O0 vs O3



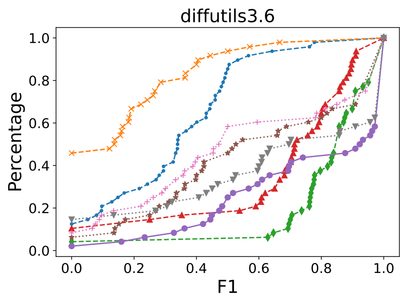
(b) O1 vs O3



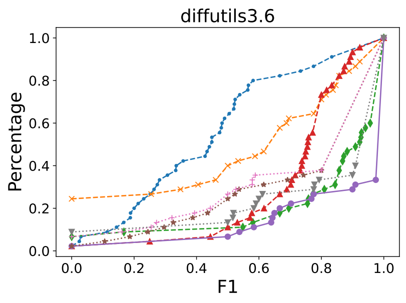
(c) O2 vs O3



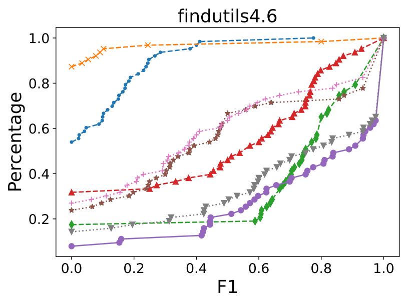
(d) O0 vs O3



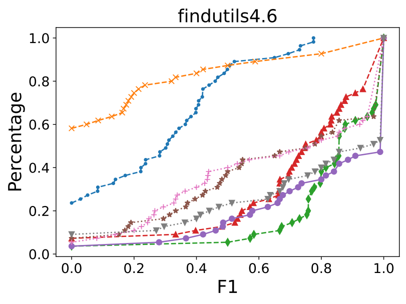
(e) O1 vs O3



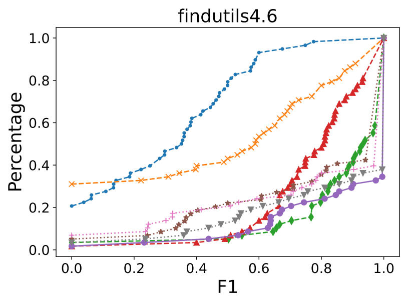
(f) O2 vs O3



(g) O0 vs O3



(h) O1 vs O3



(i) O2 vs O3

Fig. 5: Cross-optimization level diffing F1-score CDF.

to 100%. Note that Rs of IE and DBD$^{\text{IE}}$ are extremely low. The main reason is that the change of architecture largely alters the CFG, which makes it difficult to find consecutive correctly matched nodes using IE's node embeddings. In practice, they can usually only find scattered matched nodes.

5) Cross-architecture Diffing after the Compilation Environment Evolves: We simulate compilation environment evolution through source code/compiler version updates and optimization level increases. Architectures typically remain stable over extended periods to ensure backward compatibility. Specifically, we evaluate BA and three baselines on Coreutils 9.1 (GCC-8.2, O3) function pairs in the cross-architecture scenario. ML-based models in these methods are trained using Coreutils 8.30 (GCC-5.4, O0) function pairs.

As displayed in Table VI, BA maintains superior performance (achieving 49.7% $\sim$ 78% higher $F1$ than baselines). This demonstrates BA's strongest resilience to the evolution of the compilation environment compared to baseline methods. BA, SD$^{\text{G}}$, and DBD$^{\text{IE}}$ exhibit reduced performance relative to Table V. This is because the trained neural networks (BA's subgraph embedding generator, SD$^{\text{G}}$ and DBD$^{\text{IE}}$'s node matcher) cannot generalize effectively to CFG feature shifts induced by compilation environment evolution. It is interesting that IE exhibits a 15.4% $F1$ improvement. This enhancement primarily results from CFG simplification under O3 optimization: optimized CFGs have fewer basic blocks (but the single basic block becomes larger) and reduced branching. This is more conducive to IE's node matching algorithm based on linear path search, which tends to produce false positives in complex CFGs by matching non-ground-truth node pairs exceeding similarity thresholds. Nevertheless, IE's performance gains remain constrained, and BA is the optimal diffing tool after the compilation environment evolves.

TABLE V: Cross-architecture Diffing Results (x86 vs ARM)

	Approach	Coreutils 8.30	Diffutils 3.6	Findutils 4.6
R	IE	0.081	0.09	0.074
	DBD$^{\text{IE}}$	0.098	0.121	0.1
	SD$^{\text{G}}$	0.34	0.366	0.371
	BA	**0.961**	**0.935**	**0.989**
	BA-0	0.941	0.897	0.953
	BA-1	0.936	0.92	0.971
	BA-GMN	0.929	0.932	0.967
P	IE	0.44	0.47	0.352
	DBD$^{\text{IE}}$	0.322	0.326	0.204
	SD$^{\text{G}}$	0.71	0.737	0.713
	BA	**0.991**	**0.977**	**1.0**
	BA-0	0.982	0.977	0.993
	BA-1	0.985	0.97	0.993
	BA-GMN	0.954	0.977	0.98
$F1$	IE	0.137	0.151	0.122
	DBD$^{\text{IE}}$	0.15	0.176	0.134
	SD$^{\text{G}}$	0.46	0.489	0.488
	BA	**0.976**	**0.956**	**0.994**
	BA-0	0.961	0.935	0.973
	BA-1	0.96	0.944	0.982
	BA-GMN	0.941	0.954	0.973

TABLE VI: Cross-architecture Diffing Results after the Compilation Environment Evolves

	IE	DBD$^{\text{IE}}$	SD$^{\text{G}}$	BA
R	0.181	0.005	0.128	**0.688**
P	0.742	0.017	0.526	**0.921**
$F1$	0.291	0.008	0.206	**0.788**

6) Diffing on Different Function Pairs with Different Degrees of Similarity: Finally, we evaluate the performance of BA on function pairs with different degrees of similarity. DBD is not employed in this experiment due to its inability to handle large-sized binary files (e.g., *libssl.so* and *libcrypto.so*).

As shown in Table VII, BA performs significantly better than the baselines. For example, when the similarity degree of function pairs ranges from 50% to 70%, $F1$ of BA achieves an average improvement of 65.4% with a maximum gain of 85.2%. The results demonstrate that graph-level matching can achieve better binary diffing performance than node-level matching. In addition, it can be seen that the higher the similarity degree of the function pair, the better the performance of BA. This is because larger source code snippets typically correspond to larger subgraphs, which are more favorable for accurate matching.

The limited performance of the baselines is largely due to the node-level matching method. Specifically, IE's path search is restricted to semantically equivalent basic blocks, without considering any branch information of the graph. The k-hop algorithm restricts the node matching range to k hops. This restriction may lead to omissions, especially when the CFG is large. SD$^{\text{G}}$ also focuses on node matching by computing node similarities, although it uses a neural network instead of a heuristic algorithm. In contrast, BA reduces the probability of false and missed matches by proposing the conditional relaxation strategies to obtain candidate subgraph pairs. Thus, BA achieves better effectiveness.

The poor performance of IE and DBD$^{\text{IE}}$ can also be attributed to the low quality of node embeddings generated by IE. The OpenSSL dataset used to train IE embedding models comprises a blend of function pairs derived from all R_ss, ranging from 10% to 90%. Thus, it imposes a significant burden on the model that utilizes instructions as learning information. In contrast, BA employs statistical attributes to substantially reduce the learning complexity of the node feature, which is more suitable for such a complex scenario.

C. Efficiency

In this section, we evaluate the efficiency of BA. We first focus on the algorithm efficiency of graph/node matching for given function pairs. This is because the algorithm for matching subgraph pairs is a crucial component of BA, which alleviates the effectiveness limitation of existing diffing techniques. We then evaluate the overall execution efficiency of BA, considering a more realistic end-to-end binary file diffing.

1) Algorithm of Graph/Node Matching: We analyze and evaluate the algorithm efficiency of graph-level matching (i.e., BA) and node-level matching (i.e., IE and DBD$^{\text{IE}}$) in the scope of function pairs, both theoretically (i.e., time complexity) and experimentally (i.e., algorithm execution time). We represent the number of vertices (i.e., nodes) of a CFG as V, the number of edges as E, and the number of loops as L.

The algorithms used in IE mainly include loop unrolling, DFS, and LCS. The time complexity of unrolling all loops is $O((V+E)\cdot(L+1))$ [51]. The time complexity to determine the starting block and candidate starting blocks is $O(V)$, and the time complexity of using DFS to find linearly independent paths in a query CFG is $O(V\cdot(V+E))$. The time complexity of applying BFS in the target CFG, combined with the LCS dynamic programming to match basic blocks is $O(V\cdot(V+E))$.

DBD$^{\text{IE}}$ proposes the k-hop greedy matching algorithm. We mention that the initial node pairs in ICFG are changed to node pairs that exhibit maximal similarities in CFG (see Section IV-A2), so the time complexity of determining the starting node is $O(V^2)$. Neighbor node exploration incurs $O(E)$ time complexity, and sorting similarity scores of neighbor nodes requires $O(VlogV)$.

The algorithm process in BA includes initial subgraph pair identification and candidate subgraph pair obtainment (i.e., the first two steps of BA). The time complexity of our algorithm to eliminate loops is $O(V^2)$. Besides, the time complexity of identifying the initial subgraph pairs is $O(V^2)$, while expanding the subgraphs to obtain a candidate subgraph pair takes $O(V+E)$.

Table VIII lists the average matching time (in seconds) of all function pairs under four scenarios for three algorithms. In general, DBD$^{\text{IE}}$ and BA are on the same time scale, while IE is 4000~30000 times higher than DBD$^{\text{IE}}$. The matching process in IE demands a substantial amount of time, primarily due to the large number of semantically equivalent block pairs. Specifically, the increase in the number of these block pairs directly corresponds to an escalated time investment, as each pair demands an independent path search and the running of

TABLE VII: Diffing Results under Different Similarity Degree Ranges

Sim. Deg. Range (%)	Approach	R				P				$F1$			
		X(V)	X(V+C)	X(V+O)	X(V+A)	X(V)	X(V+C)	X(V+O)	X(V+A)	X(V)	X(V+C)	X(V+O)	X(V+A)
10∼30	IE	0.195	0.13	0.119	0.05	0.355	0.225	0.18	0.078	0.252	0.165	0.143	0.061
	DBD^{IE}	0.082	0.219	0.181	0.131	0.112	0.237	0.189	0.179	0.095	0.228	0.185	0.151
	SD^{G}	0.195	0.037	0.027	0.008	0.289	0.08	0.058	0.02	0.233	0.051	0.037	0.011
	BA	**0.627**	**0.504**	**0.498**	**0.512**	**0.767**	**0.678**	**0.678**	**0.715**	**0.69**	**0.578**	**0.574**	**0.597**
30∼50	IE	0.262	0.173	0.161	0.037	0.598	0.416	0.371	0.101	0.364	0.244	0.225	0.054
	DBD^{IE}	0.08	0.19	0.209	0.109	0.192	0.333	0.36	0.223	0.113	0.242	0.264	0.146
	SD^{G}	0.356	0.125	0.076	0.025	0.476	0.257	0.147	0.101	0.407	0.168	0.1	0.04
	BA	**0.867**	**0.754**	**0.647**	**0.776**	**0.959**	**0.914**	**0.853**	**0.926**	**0.911**	**0.826**	**0.736**	**0.844**
50∼70	IE	0.307	0.214	0.197	0.043	0.802	0.606	0.535	0.204	0.444	0.316	0.288	0.071
	DBD^{IE}	0.069	0.176	0.198	0.106	0.271	0.398	0.404	0.32	0.11	0.244	0.266	0.159
	SD^{G}	0.456	0.169	0.094	0.028	0.66	0.377	0.235	0.14	0.539	0.233	0.135	0.047
	BA	**0.931**	**0.851**	**0.746**	**0.853**	**0.963**	**0.935**	**0.932**	**0.95**	**0.947**	**0.891**	**0.829**	**0.899**
70∼90	IE	0.344	0.282	0.238	0.059	0.908	0.783	0.668	0.295	0.499	0.415	0.351	0.098
	DBD^{IE}	0.041	0.193	0.178	0.079	0.201	0.431	0.361	0.254	0.068	0.267	0.238	0.121
	SD^{G}	0.589	0.235	0.142	0.025	0.784	0.527	0.318	0.126	0.673	0.325	0.196	0.042
	BA	**0.964**	**0.908**	**0.833**	**0.899**	**0.974**	**0.96**	**0.936**	**0.951**	**0.969**	**0.933**	**0.882**	**0.924**

TABLE VIII: Comparison of Algorithm Matching Time

Algorithm	Matching Time (seconds per function pair)			
	Cross Vers.	Cross Opti.	Cross Comp.	Cross Arch.
IE	118.733	217.386	135.092	98.823
DBD^{IE}	**0.027**	**0.023**	**0.008**	**0.003**
BA	0.133	0.121	0.065	0.006

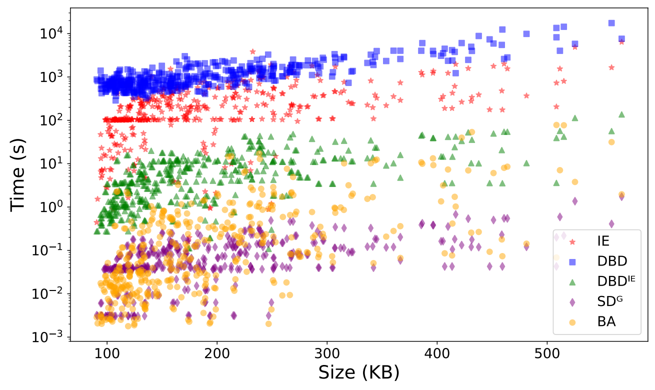


Fig. 6: Binary file (unstripped binaries taken from the cross-version scenario) size vs. execution time of different diffing approaches.

the LCS algorithm. The time consumption of DBD^{IE} is slightly less than BA, which is basically in line with our analysis of time complexity.

2) Approach of Binary Diffing: We evaluate the execution efficiency of BA and all baselines via end-to-end binary file diffing. Function names enable preliminary matching of function pairs, reflecting realistic application contexts such as plagiarism detection (e.g., student homework vs. open-source code on the Internet) where identically named functions exhibit higher matching probabilities, and vulnerability tracking where known function names guide patch/vulnerability preliminary matching.

Fig. 6 shows the execution time scaling trends of different methods across binary file sizes in the cross-version scenario. BA exhibits efficiency comparable to SD^{G} for small binary files ($<$ 200KB). Beyond this threshold, BA incurs moderately higher execution times than SD^{G}, attributable to the increased scale and complexity of CFGs in larger binary files. While this affects the matching time of the algorithm, model inference time exhibits minimal sensitivity to file size scaling. Crucially, BA, SD^{G}, and DBD^{IE} complete diffing within 100 seconds (for all binaries under 600KB), demonstrating practicality. Conversely, IE's high-complexity node matching and DBD's expensive embedding generation exhibit significant scalability limitations, with execution times increasing substantially as the file size increases (note that the y-axis is in log scale).

D. Ablation Study

In this section, we investigate the impact of different components and hyperparameters on BA with the GNU dataset. We first evaluate BA with different conditional relaxation strategies. Then, we compare different neural networks in the aligned subgraph pair decision model. Next, we evaluate the sensitivity of initial subgraph pairing to the node enumeration order and its subsequent effect on diffing performance. Finally, we set different values for the threshold of graph embeddings' similarity scores to understand the impact.

1) Conditional Relaxation Strategy: We measure the effect of different conditional relaxation strategies on binary diffing performance in BA. From Table III, Table IV, Table V and Fig. 5, we can see that the effectiveness of BA is superior to BA-0 and BA-1 in four scenarios. For example, in the cross-version scenario, $F1$ of BA is more than 14.5% higher than that of BA-0 and BA-1 on Findutils. In the cross-optimization level scenario, the CDF curve of BA consistently lies beneath the curves of both BA-0 and BA-1. Another interesting observation is that BA-1 does not always achieve better results than BA-0. For instance, in the cross-compiler scenario, $F1$s of BA-1 on Coreutils and Findutils are 1.2% and 0.8% higher than BA-0, while 1.6% lower on Diffutils. This may be because the first strategy $St.^{1st}$ is not sufficient to significantly prevent omissions. Compared with chain nodes, branch nodes may contain more information about similarity. These results illustrate that the combination of the two conditional relaxation

TABLE IX: Diffing Results of Different Node Enumeration Orders (Taken from the Cross-architecture Scenario)

	Order	Coreutils 8.30	Diffutils 3.6	Findutils 4.6
R	ascending	**0.964**	0.934	**0.989**
	descending	0.962	**0.935**	0.987
	random	0.961	**0.935**	**0.989**
P	ascending	0.991	**0.977**	**1.0**
	descending	**0.992**	**0.977**	**1.0**
	random	0.991	**0.977**	**1.0**
$F1$	ascending	**0.977**	0.955	**0.994**
	descending	**0.977**	**0.956**	0.993
	random	0.976	**0.956**	**0.994**

strategies can reduce the probability of missing graph-level information and achieve better diffing performance.

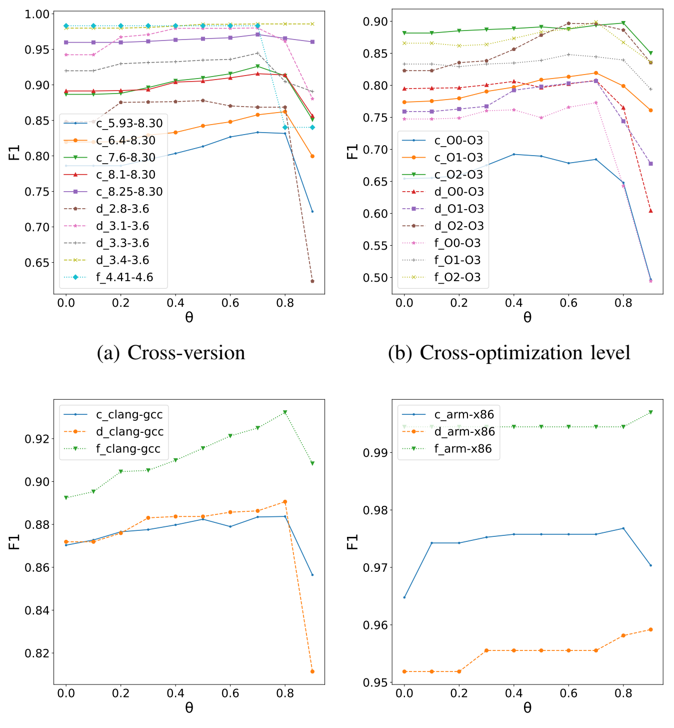
2) Aligned Subgraph Pair Decision Model: To observe the impact of subgraph embeddings generated by different neural networks on the aligned subgraph pair decision, we compare two neural networks, Gemini's graph embedding network [22] and GMN [49]. As shown in Table III, Table IV, Table V and Fig. 5, the performance of Gemini's graph embedding network is better than that of GMN. The maximum difference of $F1$ between BA and BA-GMN is 10.8% in the cross-version scenario, and the CDF curve of BA is always below that of BA-GMN. The results show that the graph embedding network learns similarity features for subgraph pairs better than GMN. The reason may be that GMN first matches each node with a similar node in another graph and then learns graph embeddings based on matched node embeddings. Essentially, GMN performs node-level matching.

3) Node Enumeration Order: In this experiment, we evaluate whether initial subgraph pairing is sensitive to the node enumeration order, thereby affecting the effectiveness of diffing. We compare three orderings in the cross-architecture scenario: ascending node number order, descending node number order, and random index order (with a random seed set to 0). The results in Table IX demonstrate that the effect of node enumeration order on the effectiveness of BA is negligible (with $F1$ difference within 0.1%), indicating that any ordering is feasible.

4) Similarity Threshold θ: We evaluate how the value of θ affects the diffing performance for cross-compilation environment scenarios of BA. The similarity score calculated by the aligned subgraph pair decision mode is a value in [0,1], so we set θ to $0.0 \sim 0.9$ with an interval of 0.1. Fig. 7 shows the variation of $F1$ concerning θ in four scenarios. We can see that most of the $F1$ change curves exhibit an initial increase followed by a decrease as θ increases. This is because initially increasing θ allows for the model to exclude incorrectly aligned candidate subgraph pairs. However, when θ becomes excessively large, correctly aligned candidate subgraph pairs may also be discarded. Therefore, we conservatively choose 0.7 as the value of θ.

E. Case Study

We further exhibit BA's utility to real-world vulnerability/patch analysis through two case studies: the first uses libraries and compilation environments within the model's training distribution, while the second employs previously unseen libraries and compilation environments.



(a) Cross-version (b) Cross-optimization level

(c) Cross-compiler (d) Cross-architecture

Fig. 7: Impact of different similarity threshold θ.

TABLE X: Diffing Results for Heartbleed on OpenSSL

Approach	Vulnerability			Patch		
	X(A)	X(A+O)	X(A+O+C)	X(A)	X(A+O)	X(A+O+C)
IE	0.0	0.0	0.0	0.0	0.0	0.0
DBD^{IE}	**1.0**	**1.0**	0.5	0.5	0.667	0.375
SD^{G}	**1.0**	0.0	0.0	0.75	0.0	0.417
BA	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**

1) Practical Application: Heartbleed (CVE-2014-0160) is a classic vulnerability in OpenSSL that exists in version 1.0.1f and gets fixed in version 1.0.1u. We identify the vulnerability and the patch in these two versions for an in-depth comparison between BA and the state-of-the-art diffing tools. Table X displays diffing results (we use the recall score R) in three cross-compilation environment scenarios. The first scenario is (x86-64, O0, GCC-8.2.0) vs (ARM-64, O0, GCC-8.2.0), i.e., cross-architecture, represented as X(A). The second scenario is (x86-64, O0, GCC-8.2.0) vs (ARM-64, O3, GCC-8.2.0), i.e., cross-architecture and cross-optimization level, expressed as X(A+O). The third scenario is (x86-64, O0, GCC-8.2.0) vs (ARM-64, O3, Clang-6.0), i.e., cross-architecture, cross-optimization level, and cross-compiler, indicated as X(A+O+C). As we can see, BA outperforms all three baselines, demonstrating its utility for pinpointing the vulnerability/patch in complex cross-compilation scenarios. Furthermore, IE's zero recall ($R = 0$) underscores the importance of considering branch information, particularly when analyzing targets containing low-optimization-level CFGs.

2) Robustness Verification: To validate BA's robustness in real-world applications, we conduct this case study on nine vulnerabilities and patches from five libraries (Binutils, Tar, Libmicrohttpd, Inetutils, Recutils) compiled with Clang-13/O3 for cross-architecture (x86-64 vs ARM-64) diffing. Models are trained on Coreutils/Diffutils/Findutils compiled under GCC-5.4/O0. Table XI details these vulnerabilities and patches, and the evaluation results of four diffing methods. We confirm vulnerability/patch presence in a function when corresponding basic blocks exist in the matching results (i.e., $R > 0$). Overall, the robustness of these four methods is similar to the results in Section IV-B5. BA achieves 100% success (9/9) despite encountering unseen libraries, functions, and the compilation environment. IE and SD^{G} perform suboptimally (7/9 and 5/9, respectively), attributed to the node matching algorithm suitable for high optimization levels (IE) and the deep neural network-based node matching model (SD^{G}). DBD^{IE} fails completely (0/9) due to limited model generalization capability. IE's node embedding generation model cannot produce high-similarity embeddings for unseen basic blocks. The results highlight the robustness of BA for practical applications.

V. Discussion

In this section, we discuss some limitations of BinAligner and future work.

Semantic Features. To achieve a flexible cross-compilation environment diffing at the binary level, BinAligner employs basic block statistical attributes in Genius for graph embeddings, which avoid architectural constraints inherent to instruction syntax. However, this approach limits the exploitation of instruction information. Future work will abstract instructions to the IR level, thereby integrating natural language processing techniques for enhanced semantic analysis.

Attack Robustness. BinAligner can partially handle code optimization (e.g., function inlining via simple C macros) and obfuscation (e.g., basic opaque predicates) problems. Specifically, single-block changes like constant hiding or expression obfuscation do not affect subgraph pairing. Chain/simple branch modifications with preserved topology (e.g., opaque predicates or basic block splitting) are addressable. However, topology-altering transformations such as control flow flattening or loop nesting may cause partial match failures, exposing attack surfaces where sophisticated branch alterations could challenge BinAligner.

Disassembler Difference. Our implementation utilizes IDA Pro 7.5 [42]. It should be noted that different disassembly tools (e.g., Ghidra [52]) or versions (e.g., IDA Pro 6.5) may produce different CFGs. Consequently, applying BinAligner to these variants could introduce minor experimental biases. Nevertheless, as a mainstream disassembler with broad adoption in state-of-the-art methods, the use of IDA Pro is sufficient to prove BinAligner's practicality.

Indirect Jumps. BinAligner relies on the disassembly tool for indirect jump branch recovery. We ignore indirect jumps when dividing the chain/anchor node, which may influence diffing results. Nevertheless, this limitation exists in all diffing methods. We leave this issue for future work.

Ground Truth of Source Code. We conservatively employ textual equivalence as the ground truth for the same source codes, which is sound but incomplete. Our future work aims to incorporate source code similarity detection to capture more functionally identical but syntactically different transformations that the current ground truth may not fully identify.

Scalability. For one given function pair, BinAligner performs binary diffing once. Future work should consider the problem of function retrieval for large binaries (e.g., containing hundreds of thousands of functions). In addition, our current datasets (i.e., GNU and OpenSSL) come from dynamically linked binaries, so different functions are easy to separate. In contrast, function boundaries for statically linked binaries may not be well-defined, which may adversely influence the diffing performance of BinAligner.

Loop Elimination. Removing an edge may cause an anchor node to become a chain node or vice versa. This may induce bidirectional incorporation: incorporating the node that is originally excluded or excluding the node that is originally incorporated. Consequently, the positive and negative impact of loop elimination on BinAligner's performance may remain balanced. Further exploration is deferred to future work.

VI. Related Work

In this section, we introduce traditional binary and pseudocode diffing approaches (the ML-based methods are introduced in Section II-C).

Traditional binary diffing approaches usually match basic blocks based on fixed rules and static syntax features. BinDiff [1] establishes basic block correspondences through various features such as instruction prime products, hashes of raw bytes, call references, string references, etc. Binslayer [27] combines BinDiff with the Hungarian algorithm for bipartite node matching. Dullien et al. [30] iteratively refine call graph isomorphisms via hierarchical function/basic block/instruction matching, while BinHunt [2] achieves subgraph isomorphism through symbolic execution-based basic block similarity comparison. CoP [29] similarly employs symbolic execution to verify basic block semantic equivalence and identify the longest semantically equivalent subsequences. Pewny et.al. [28] compute basic block I/O pair hashes for bug discovery, contrasting with Tracelet's [31] path-based function decomposition and register/memory matching. Esh [32] decomposes code into smaller comparable fragments and verifies intermediate/output value equivalence, and GitZ [33] is a reoptimized version. Falleri et al. [53] compare code differences in the abstract syntax tree. Traditional methods have limited effectiveness, especially in cross-compilation environment scenarios.

Traditional pseudocode diffing tool Diaphora [54] is driven by simple heuristic algorithms. It treats two code snippets as two strings and conducts string-based matching to diff the two code snippets. Therefore, it has little robustness to pseudocode changes caused by changes in the compilation environment.

TABLE XI: List of CVE Vulnerabilities and Patches Detected on Other Open-source Libraries

Library	Ver.	CVE	Vul./Pat.	Function	IE	DBD^{IE}	SD^{G}	BA
Binutils	2.4	CVE-2025-5245	Vul.	**debug_type_samep**				✓
		CVE-2025-5244	Vul.	**bfd_elf_gc_sections**	✓		✓	✓
		CVE-2025-1176	Vul.	**_bfd_elf_gc_mark_rsec**	✓		✓	✓
Tar	1.34	CVE-2023-39804	Vul.	**xheader_decode**	✓		✓	✓
		CVE-2023-39804	Vul.	**xattr_decoder**	✓		✓	✓
Libmicrohttpd	0.9.75	CVE-2023-27371	Vul.	**MHD_create_post_processor**	✓			✓
Inetutils	2.4	CVE-2022-39028	Pat.	**telrcv**	✓			✓
		CVE-2021-40491	Pat.	**initconn**			✓	✓
Recutils	1.9	CVE-2021-46022, CVE-2021-46019	Pat.	**rec_parse_comment**	✓			✓

VII. Conclusion

In this paper, we propose a novel binary diffing approach BinAligner to alleviate the limitations of existing works in effectiveness and flexibility across compilation environments. To improve effectiveness, we propose graph-level matching to obtain candidate subgraph pairs. To increase flexibility at the binary level, we use instruction-independent basic block features to generate subgraph embeddings. We evaluate BinAligner in terms of the effectiveness, support ability for different scenarios, and compilation environment evolution resilience, as well as utility for real-world vulnerabilities and patches. Our experimental results demonstrate that the proposed approach outperforms state-of-the-art methods.

Acknowledgment

We are deeply grateful to the anonymous reviewers for their valuable comments, which greatly enhanced the quality of our paper. We also thank Sijie Zhi, Qiuyu Li, and Kecheng Li for their contributions to the preliminary research of the paper. This work is supported by ZJU Kunpeng&Ascend Center of Excellence. We appreciate the support of Kunpeng&Ascend server computing resources.

References

[1] "zynamics BinDiff," https://www.zynamics.com/bindiff.html, 2023. 1, 16

[2] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20-22, 2008 Proceedings 10*. Springer, 2008, pp. 238–255. 1, 16

[3] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and distributed system security symposium*, 2019. 1, 4, 9

[4] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and distributed system security symposium*, 2020. 1, 4, 9, 10

[5] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 868–901, 2010. 1

[6] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008. 1

[7] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017. 1

[8] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 117–128. 1

[9] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472. 1

[10] Y. Li, W. Xu, Y. Tang, X. Mi, and B. Wang, "Semhunt: Identifying vulnerability type with double validation in binary code." in *SEKE*, 2017, pp. 491–494. 1

[11] J. F. Islam, M. Mondal, and C. K. Roy, "Bug replication in code clones: An empirical study," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 68–78. 1

[12] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95. 1

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105. 1

[14] G. Tao, D. Guowei, Q. Hu, and C. Baojiang, "Improved plagiarism detection algorithm based on abstract syntax tree," in *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*. IEEE, 2013, pp. 714–719. 1

[15] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678. 1

[16] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla *et al.*, "discovre: Efficient cross-architecture identification of bugs in binary code." in *Ndss*, vol. 52, 2016, pp. 58–79. 1

[17] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 78–87. 1

[18] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 329–338. 1

[19] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116. 1

[20] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489. 1

[21] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020. 1

[22] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376. 1, 2, 4, 7, 9, 15

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491. 1, 2, 4, 7

[24] G. Kim, S. Hong, M. Franz, and D. Song, "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*, 2022, pp. 151–163. 1

[25] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2409–2426. 1

[26] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023. 1

[27] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013, pp. 1–10. 1, 16

[28] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 709–724. 1, 16

[29] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 389–400. 1, 16

[30] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *Sstic*, vol. 5, no. 1, p. 3, 2005. 1, 16

[31] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014. 1, 16

[32] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *Acm sigplan notices*, vol. 51, no. 6, pp. 266–280, 2016. 1, 16

[33] ——, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 79–94. 1, 16

[34] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward, "Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval," *IEEE/ACM transactions on audio, speech, and language processing*, vol. 24, no. 4, pp. 694–707, 2016. 1, 4

[35] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, vol. 2015, 2015, pp. 2111–2117. 1, 4

[36] L. GAO, Y. QU, S. YU, Y. DUAN, and H. YIN, "Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing," in *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS 2024), San Diego, CA, USA, February*, 2024, pp. 1–19. 2, 4, 9, 11

[37] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege, "Deep graph matching consensus," *arXiv preprint arXiv:2001.09621*, 2020. 2, 4

[38] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993. 2, 4

[39] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*. PMLR, 2016, pp. 2702–2711. 2, 4, 7

[40] "OpenSSL," https://www.openssl.org/, 2012. 3, 8

[41] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013. 4

[42] H. Rays, "IDA Pro," https://www.hex-rays.com/products/ida/, 2019. 4, 7, 16

[43] I. Jane Bromley, I. Guyon, and R. Shah, "Signature verification using a" siamese."," *Time Delay Neural Network International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, p. 25, 1993. 7

[44] "Gnu coreutils," https://www.gnu.org/software/coreutils/, 2024. 8

[45] "Gnu diffutils," https://www.gnu.org/software/diffutils/, 2024. 8

[46] "Gnu findutils," https://www.gnu.org/software/findutils/, 2024. 8

[47] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "Sok: Benchmarking flaws in systems security," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 310–325. 8

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. 9

[49] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845. 9, 15

[50] Python Software Foundation, "difflib – helpers for computing deltas," https://docs.python.org/3/library/difflib.html, Python 3.7. 10

[51] "networkx.algorithms.cycles.simple_cycles," https://networkx.org/documentation/networkx-2.4/reference/algorithms/generated/networkx.algorithms.cycles.simple_cycles.html, 2023. 13

[52] N. S. Agency, "Ghidra reverse engineering tool," https://ghidra-sre.org/, 2019. 16

[53] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324. 16

[54] "Diaphora," https://github.com/joxeankoret/diaphora, 2024. 16