Artifact Evaluated NDSS
Available
Functional
Reproduced

Enhancing Semantic-Aware Binary Diffing with High-Confidence Dynamic Instruction Alignment

Chengfeng Ye, Anshunkang Zhou✉, Charles Zhang
The Hong Kong University of Science and Technology, China
{cyeaa, azhouad, charlesz}@cse.ust.hk

***Abstract*—Binary diffing, which detects differences between two pieces of binary code, is the fundamental technique in various security analysis tasks. Existing work shows that a sufficient number of fine-grained alignments as anchor points can significantly improve the overall accuracy of binary diffing. However, existing methods still suffer from numerous limitations that hinder accurate and efficient anchor point identification. Syntax-based techniques are known to be vulnerable to aggressive compiler optimizations, while semantic-based methods are limited by high computation cost or low code coverage.**

In this paper, we revisit dynamic analysis to seek new insights to address the limitations of existing approaches. Our main insight is that not all dynamic semantics are necessary or equally effective for identifying valid instruction alignment. Therefore, we can prioritize dynamic execution resources to partially reveal the runtime values that can effectively derive instruction alignment. Based on the above insight, we propose BARRACUDA, a high-confidence instruction alignment technique based on partial instruction semantics extracted from forced execution. We have implemented BARRACUDA and conducted extensive experiments to evaluate its effectiveness. Extensive experimental results demonstrate that BARRACUDA can detect 24.0% more instruction alignment as anchor points with a high precision of 92.1%. The anchor points detected by BARRACUDA can enhance state-of-the-art binary diffing tools, DEEPBINDIFF and SIGMADIFF, with percentage point increases in F1 scores ranging from 12.3% to 42.7% and 2.2% to 4.1%, respectively, across various binary diffing scenarios.

I. INTRODUCTION

Binary diffing identifies the differences or modifications between two pieces of binary code. This fundamental technique facilitates a range of downstream binary security analyses, including one-day vulnerability detection [1]–[5], plagiarism detection [6], [7], code change analysis [8]–[10], malware analysis [11], supply chain analysis [12], [13] and patch presence analysis [14]–[16].

The basic idea behind binary diffing is to match code parts that are found to be equivalent or similar and take the rest of them as differences. Over the past few decades, there have been significant advancements in this area, ranging from coarse-grained functional-level methods [17]–[28] to finer-grained basic-block-level [8], [29]–[34] and instruction-level techniques [9], [10], [35]. Although existing efforts focus on matching code fragments at different granularities, one of the most important steps, and usually the very first one, is called "anchor points identification". Simply speaking, anchor points are pairs of similar or equivalent code fragments between the compared binaries that can be matched together with high confidence. By identifying anchor points in the first place, previous methods could perform subsequent fuzzy matching stages to match the remaining code effectively and find the differences. Existing practices [10], [33] have shown that a sufficient number of pre-existing anchor points can significantly improve the overall binary diffing accuracy, and the greater the number of high-confidence matches known in advance, the more accurate the final diffing result will be. For example, SIGMADIFF [10] requires adequate pre-known matches to serve as labeled training nodes before its semi-supervised deep graph matching [36], and the accuracy of DEEPBINDIFF [33] can be improved by connecting pre-matched nodes on the merged inter-procedural control flow graphs (ICFGs) of two compared binaries before performing k-hop greedy matching.

Despite the importance of accurate anchor point identification, we found that existing work still falls short in finding an adequate number of anchor points due to several limitations.

Limitations of Existing Efforts. First, some existing binary diffing techniques heavily rely on syntactic signatures such as control-flow graphs [8] and instruction sequences [9] to find matched anchor points. However, merely leveraging syntactic signatures is known to be vulnerable to aggressive compiler optimizations [33], [37]. For example, although SIGMADIFF [10] tries to define the signature as symbolic formulas of instructions, they still choose to determine the equivalence of two symbolic formulas by comparing their syntactic structure, which can be easily changed by compiler optimizations [26].

Second, another line of work leverages semantic-based methods that extract binary functionalities or behaviors through dynamic execution [34] or symbolic execution [14], [15], [30], [31] for matching equivalent code fragments. On the one hand, symbolic execution still could cause scalability issues. Therefore, binary diffing techniques that rely on symbolic execution either take a long time (from three to six hours) just to find basic-block matching in small binaries with only

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.2026.240663
www.ndss-symposium.org

thousands of source lines [30], [31], or can at most be applied on restricted code region to perform specific downstream tasks like patch-presence testing [14]–[16].

On the other hand, although dynamic approaches have the advantage of efficiently deriving robust binary program semantics from runtime values observed during concrete execution [35], [38], [39], they naturally suffer from low coverage due to path explosion. Forced execution-based methods [37], [40], [41] try to increase block coverage by overriding the intended program logic. While semantics extracted through high block coverage is sufficient for analyzing function-level similarity [37], they are still partial ones, as with finite computational resources, it is always difficult to cover all execution paths within a binary. Therefore, existing methods still fall short when analyzing finer-grained instruction alignment.

Insight. In this paper, we revisit dynamic analysis to uncover new insights to address the aforementioned limitations. Our key insight is that not all dynamic semantics are necessary or equally effective for identifying valid anchor points. On the one hand, drawing on the successful use of biclique structures in bioinformatics [42] and recommendation systems [42], we notice that a group of instructions with partially overlapping semantic relationships forming a biclique strongly indicates the presence of valid alignment within that group. On the other hand, it is intuitive that two instructions from the compared binaries are more likely to share overlapping dynamic semantics if they operate on a smaller set of runtime values, and revealing just a small portion of these values is sufficient to achieve a high probability of overlap. Based on these observations, we could prioritize the execution resources to cover instruction values that are more likely to derive overlapping semantics and resort to the biclique structure of overlapping relationships to find instruction alignment.

Our Approach. Based on the above insight, we propose BARRACUDA, a high-confidence instruction alignment technique based on partial instruction semantics extracted from dynamic execution. To effectively derive dynamic semantics and identify more initial matches, we prioritize execution resources to reveal runtime values that are likely to produce overlapping semantics between instructions. The prioritized path sampling begins by estimating the number of distinct runtime values for each instruction through static analysis. We then start with instructions with a smaller estimated number of runtime values and sample each path that covers their symbolic formulas. This process continues until we reach a point where the probability of overlapping semantics with their counterparts in the reference binary is sufficiently high.

With only partial semantics available, we model binary instruction alignment as a two-stage bipartite graph-matching process. Two instructions are marked as an initial match as long as their semantics overlap. Then, for each balanced biclique formed from this initial matching, a small reduced CFG is constructed on each side by removing irrelevant instructions while preserving the original topological order of the remaining instructions. Deterministic graph isomorphism can finally be used to efficiently determine precise instruction matchings, leveraging the small size of the reduced graph. In this way, we can find alignment for a large portion of instructions while maintaining high overall precision.

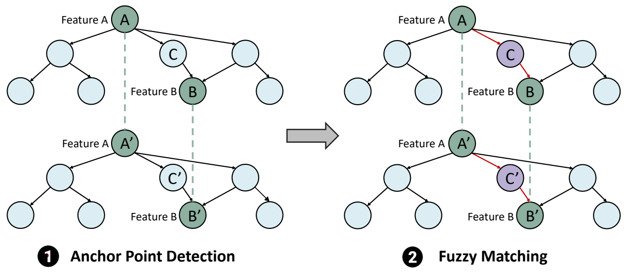


Fig. 1: Conceptual flow of modern binary diffing.

The instruction alignment results can be integrated into existing binary diffing tools as anchor points, enhancing the overall accuracy of the binary diffing process.

Evaluation. We have implemented BARRACUDA on the LLVM framework [43] and conducted extensive experiments to evaluate its effectiveness. Extensive experimental results demonstrate that BARRACUDA can detect 24.0% more instruction alignment as anchor points with a high precision of 92.1%. The anchor points detected by BARRACUDA can enhance state-of-the-art binary diffing tools, DEEPBINDIFF and SIGMADIFF, with percentage point increases in F1 scores ranging from 12.3% to 42.7% and 2.2% to 4.1%, respectively, across various binary diffing scenarios.

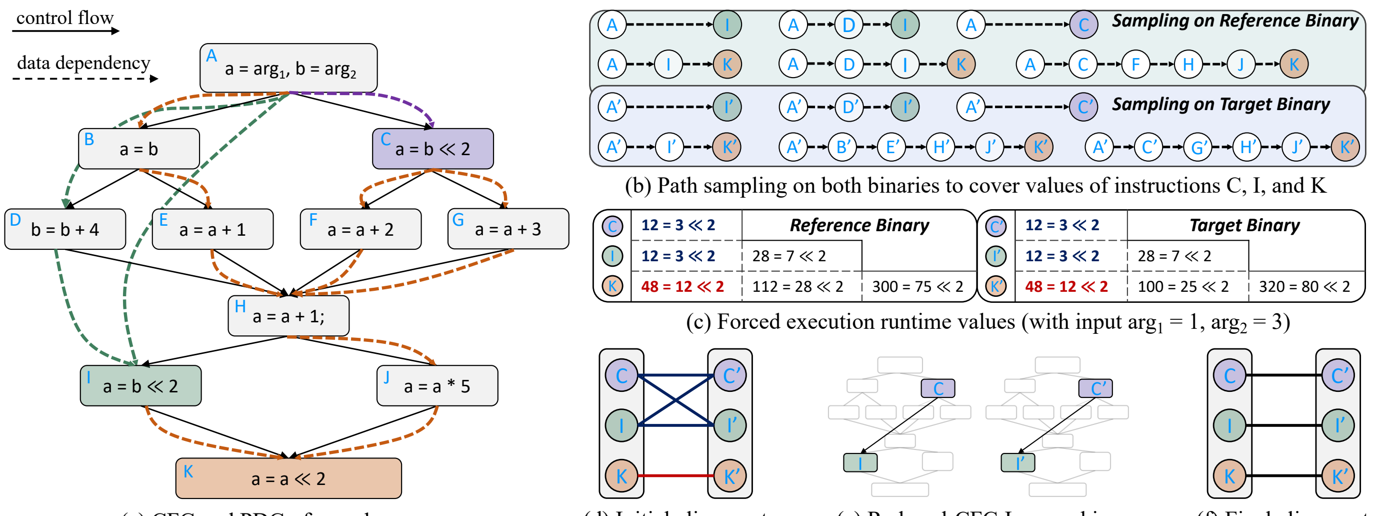
Contributions. We make the following contributions.

- *Prioritized Path Sampling:* We propose a prioritized path sampling to effectively reveal runtime values that are more likely to derive overlapping semantics on matched instructions.
- *High-Confidence Instruction Alignment:* We propose an efficient and high-confidence instruction alignment using Reduced-CFG Isomorphism on balanced bicliques constructed from overlapping semantics between instructions.
- *Extensive Evaluation:* Our evaluation demonstrates that BARRACUDA is effective at identifying more precise anchor points, thereby improving the overall diffing accuracy of the two latest binary diffing tools.

II. BACKGROUND AND MOTIVATION

A. Binary Diffing with Neighborhood Consensus

Binary diffing is commonly modeled as a graph matching problem, where instructions or basic blocks are represented as nodes in a control flow graph (CFG) or a data dependency graph (DDG). Modern binary diffing approaches [10], [33] incorporate an important concept from the graph matching domain known as Neighborhood Consensus [36], [44], [45], which considers the nodes adjacent to a given node when determining its match. Typically, these diffing methods begin with an initial matching stage to identify high-confidence matches that serve as anchor points, so that neighbors of the anchor points are more likely to be matched in the subsequent fuzzy matching stage. For instance, in Figure 1, if node pairs



(a) CFG and PDG of sample program (b) Path sampling on both binaries to cover values of instructions C, I, and K (c) Forced execution runtime values (with input $\text{arg}_1 = 1$, $\text{arg}_2 = 3$) (d) Initial alignment (e) Reduced-CFG Isomorphism (f) Final alignment

Fig. 2: This figure shows the control flow graph (CFG) and data dependency graph (DDG) of a sample program, along with our method for performing instruction alignment. Our path sampling first efficiently samples a small number of paths to reveal the instruction semantics of C, I, and K via forced execution and identifies their initial matching. With initial matching containing multiple targets, our reduced-CFG isomorphism can then efficiently and precisely detect the final matching.

A-A' and B-B' are identified as anchor points due to shared features, then node pair C-C' is more likely to be matched by subsequent fuzzy matching. In essence, the greater the number of precise early matches, the more likely the surrounding nodes will also be matched, resulting in improved overall accuracy.

B. *Forced Execution for Binary Diffing*

Forced execution is a technique designed to address the limited coverage issue in dynamic analysis. Its basic idea is that the jumping direction at each branch instruction is determined by a path-sampling algorithm rather than the runtime value of the branch condition. In a typical workflow of forced execution for binary similarity analysis, it initializes a fixed external environment (e.g., values of function parameters) for a function, then explores the program paths within the function and takes observed runtime values as the function semantics. In this way, there is no need for test cases or architecture-specific runtime environments to drive the execution.

However, because forced execution is still a per-path analysis technique, it suffers from limited coverage, as the number of program paths can grow exponentially with program size. Consequently, existing forced execution techniques [37], [40], [41] only aim to achieve full block coverage, regardless of the coverage of different paths. Moreover, these techniques only perform function-level matching rather than fine-grained instruction alignment.

III. BARRACUDA IN A NUTSHELL

A. *Motivating Example*

In this section, we use a running example to illustrate how BARRACUDA effectively samples paths to derive runtime values as instruction semantics with forced execution and utilizes the partial semantics obtained to align instructions.

Figure 2a shows the control flow graph of a code snippet, with dashed lines highlighting the data dependencies between instructions. Consider that this code snippet has been compiled into reference and target binaries. To find instruction alignment, BARRACUDA samples execution paths in both binaries to obtain runtime values of instruction operands, and considers two instructions with overlapping values as initial matches. For simplicity, we will focus on describing matching three left-shift operators: C, I, and K.

For the first point, we note that the difficulty in matching these three instructions is different. Examining the data dependencies of instruction C and I reveals that only one execution path is required for C, and two paths are required for I, to reveal all their runtime values. Specifically, the data-dependency path required for C is $A \rightarrow C$, and paths required for I are $A \rightarrow I$ and $A \rightarrow D \rightarrow I$. In contrast, instruction K requires a total of five different paths to reveal all its values, as there are five distinct data-dependency paths leading to it. Therefore, if the path sampling quota is limited, it is more beneficial to sample paths that can reveal the values of I and C rather than those of K. For the second point, we emphasize that an instruction inside two binaries can match as long as the runtime values overlap. Consequently, finding a correct match for an instruction does not require revealing all its values, and the benefit of revealing more values decreases as we reveal more values on it. Regarding instruction K, as long as we can reveal three of its five runtime values in both binaries, the two sets of values should overlap. Further sampling to reveal its runtime values would not lead to additional gain in finding more alignment. Based on the above analysis, BARRACUDA decides to cover the paths shown in Figure 2b to match all three left-shift operators, including all the paths for I and C, and three paths for K. It is important to note that these

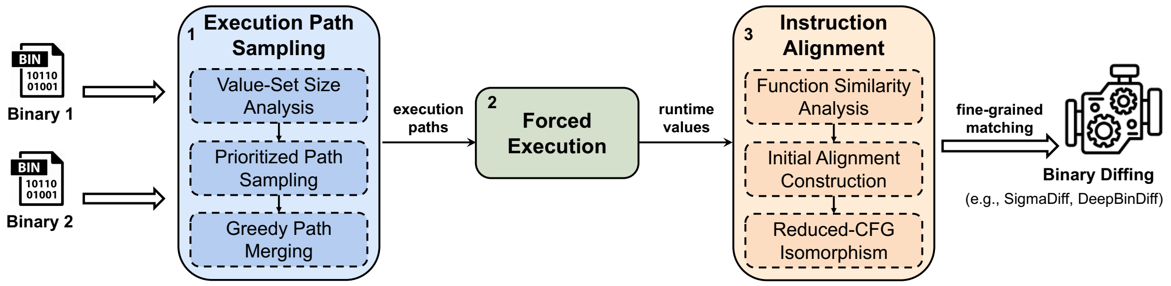


Fig. 3: Workflow of BARRACUDA

paths represent data-dependency sequences that should be covered rather than final control-flow paths to execute. For example, to cover the five selected data-dependency paths in the reference binary, executing just three complete control-flow paths is sufficient. These paths are $A \to B \to D \to H \to I \to K$, $A \to C \to F \to H \to I \to K$, and $A \to C \to F \to H \to J \to K$. Figure 2c shows the runtime values revealed by executing paths that can cover the sampled data dependencies. Based on these values, the initial matchings shown in Figure 2d can be established.

The initial alignment of an instruction could include multiple targets. For instance, as shown in Figure 2d, the targets for instruction C and I include both C' and I' because they share the same runtime value $12 = 3 \ll 2$. To precisely identify which instruction aligns with which target within an alignment group, our method is based on the assumption that the control-flow order between two instructions is generally stable against compiler optimizations. Consequently, we generalize the process of finding one-to-one instruction alignments among a group of initially aligned instructions as a graph isomorphism problem on a reduced graph that reflects control-flow order between instructions to align. As shown in Figure 2e, we construct the reduced graph for C and I by reducing the original CFG to a subgraph that excludes other unrelated instructions. Given the small scale of the reduced CFGs, a simple graph isomorphism analysis with depth-first search (DFS) can efficiently determine precise matching relationships, ultimately identifying that C matches C' and I matches I', as shown in Figure 2f. Unlike some previous approaches that apply graph isomorphism to the entire CFG [8], [9], reduced-CFG isomorphism has the advantages in both efficiency and accuracy. Since each reduced CFG contains only a small number of instructions, even a non-polynomial isomorphism analysis can complete its execution quickly. Additionally, because only the instructions to be aligned exist in the reduced CFG, code transformations performed in other code regions by compilers will not affect the alignment of instructions in a reduced CFG.

B. Workflow

In this section, we describe the overall workflow of BARRACUDA. Figure 3 shows the architecture of BARRACUDA, consisting of three main stages.

S1: Execution Path Sampling(§V-A). To sample paths that effectively reveal instruction semantics for instruction alignment, our approach is based on two key insights: (1) instructions with a smaller set of runtime values are easier to have overlapping values with their correct matching targets, and (2) the gain from revealing additional runtime values for aligning an instruction diminishes as more values on the instruction are revealed. To begin, we estimate the number of distinct runtime values of each instruction by the number of symbolic formulas that can be derived from it, which we will refer to as *value-set size*. Next, for each instruction inside a function with an increasing value-set size, we sample paths to cover its symbolic formula one by one until the probability of its value-set overlap between two binaries exceeds a large predefined threshold, or until we reach a maximum path sampling bound for a function. Finally, a greedy algorithm is applied to merge these paths into a smaller set and convert them into control-flow paths for execution.

S2: Forced Execution(§V-B). Given a path to execute, per-function forced execution is performed by initializing function parameters with a fixed set of values and then executing the function in the path order. Runtime values observed on the instruction operands are collected as instruction semantics, while the values stored in memory or passed to library functions are collected as function semantics.

S3: Instruction Alignment(§VI). Following previous work [38], [40], we first calculate pairwise function-similarity scores using the Jaccard index based on function semantics. In each iteration, we select a pair of functions with the highest score and establish an initial instruction alignment based on the overlap of instruction semantics collected from forced execution. Next, for each group of instructions whose matches form a balanced biclique, we construct a reduced CFG that includes only those instructions while preserving their relative control-flow orders. We then perform graph isomorphism on the reduced CFG to determine the precise one-to-one instruction alignment relationship within the group. Additionally, basic-block matching is derived from the alignment result of instructions within each basic block.

The output of BARRACUDA includes precise instruction alignments and basic-block matches, each marked by its binary address. These outputs are fed into existing semantic-aware binary diffing tools and serve as anchor points, enhancing their overall diffing accuracy.

$$
\begin{aligned}
\text{Program } P &:= F^+ \; G^+ \\
\text{Function } F &:= f(arg_1, ..., arg_n)\{B^+\} \\
\text{Basic Block } B &:= I^+ \\
\text{Instruction } I &:= v_0 \leftarrow \oplus(v_1, ..., v_n) \\
\text{Opcode } \oplus &:= \text{load} \mid \text{store} \mid \text{call} \mid \text{cmp} \mid \text{phi} \mid + \mid - \; ... \\
\text{Value } v &:= G \mid Arg \mid F \mid B \mid I \mid C \\
\text{Argument } arg \in Arg \quad & \text{Global } g \in G \quad \text{Constant } c \in C
\end{aligned}
$$

Fig. 4: Program Language Syntax

IV. Preliminary

In this section, we describe the definitions used in the paper and formulate the problems we aim to solve.

Language Syntax. Figure 4 illustrates the language syntax of the intermediate language (IR) on which our analysis is performed. Each binary program consists of a set of functions *F* and global variables *G*, which are recovered through disassembly [46] and decompilation [47]. Each function contains a set of basic blocks *B* arranged in control-flow order, where each low-level instruction is translated to an SSA instruction *I* with different opcodes $\oplus$. To ensure scalability in our analysis, we follow the practice from existing forced execution [41] to break back edges in the CFGs and call graph. Each function is transformed into a single-entry single-exit (SESE) structure.

Forced Execution. Following existing work [40], [41], forced execution is conducted as a per-function analysis. Figure 5 shows the abstract domain related to the forced execution. Given a fixed set of values initialized as input inp for the function parameters and a path p to be executed, forced execution interprets each instruction in the order specified by the execution path while maintaining a mapping from each top-level value v in IR and abstract memory address a to a concrete runtime value rv. In line with the memory model used by previous work [48]–[51], we organize the abstract memory into a global region R_G, a stack region R_F for each function, and a heap region R_H for each heap allocation.

Next, we formally define the concept of instruction semantics based on the runtime values observed during forced execution.

Definition 1. *The semantics of an instruction is defined as* $\{(\oplus,\, S(v_0),\, S(v_1),\, ...,\, S(v_n)) \mid inp,\, p\}$*, where* inp *is the initialized values of arguments and p is the path to be executed.*

In other words, the semantics of an instruction is represented by the set of its operand runtime values $S(v_i)$ observed during forced execution, along with its opcode $\oplus$. Since the input inp is fixed for each function to ensure the semantics derived in different functions across two binaries are comparable, the instruction semantics extracted is solely determined by the set of execution paths p.

Based on the above formal definition, we can now formulate the specific technical problems we aim to address before delving into the technical details in the following sections.

Input $inp :=\ arg \rightarrow rv$	Path $p := B^+$
Memory $M := a \rightarrow rv$	State $S := v \rightarrow rv$
Address $a := (rgn,\ \mathbb{Z}^+)$	Runtime Value $rv := \mathbb{Z}^+$
Region $rgn := R_G$ (global) $\mid {R_H}^+$(heaps) $\mid {R_F}^+$(stacks)	

Fig. 5: Forced Execution Abstract Domain

Problem Statement.

1) Given that computational resources are limited and the number of paths is exponential in relation to the size of the binary, how can we effectively sample execution paths to derive instruction semantics for alignment?
2) Given that instruction semantics is often partial due to the difficulty of enumerating all paths within a program, how can we leverage partially collected instruction semantics to match as many instructions as possible precisely?

V. Instruction Semantics Extraction

A. Execution Path Sampling

This stage samples paths based on the instruction value set size. Path sampling is conducted as a per-function analysis, where each sampled path represents a sequence of basic blocks from the entry to the exit block of the currently analyzed function f. Due to compiler optimization, some functions may be inlined in only one of the two compared binaries, making it challenging to align instructions within these functions. To address the challenge, we use strategies from Asm2Vec [22] to infer call sites that could be inlined if compiler optimization were applied, which we refer to as inlined calls. Path sampling can then proceed into the inlined callee when it encounters an inlined call. Other call sites are treated like external functions.

1) Value-Set Size Analysis: The value-set size of an instruction is the number of its distinct symbolic formulas. For an instruction $I := v_0 \leftarrow \oplus(v_1, ..., v_n)$, the value-set size can be calculated based on the instruction type and its operands:

$$
\begin{aligned}
&Size(I) = 1 && \oplus \in \{G, Arg, F, B, C\} \\
&Size(I) = \sum_{i=1}^{n} Size(v_i) && \oplus \in \{phi\} \\
&Size(I) = \sum_{v_k \rightarrow I} Size(v_k) && \oplus \in \{load\} \\
&Size(I) = Size(v_{ret}) && \oplus \in \{inlined\ call\} \\
&Size(I) = \prod_{i=1}^{n} Size(v_i) && \oplus \in \{binop, cmp, call, store\}
\end{aligned}
$$

As the formulas indicate, the value-set size for constant values, addresses, and base function arguments is set to 1. The value-set size of a phi instruction is the sum of the value-set sizes of its incoming values. A special case arises for load instruction, whose value-set size is derived from values that can be loaded from memory, represented as $v_k \rightarrow I$, obtained from a data dependency analysis. For an inlined call, we track

Algorithm 1: Control-Flow Path Conversion

```
Input: Symbolic formula $s$
Output: Control-flow path $p$ to cover the fomula $s$
1  Function ConvertToPath($s$):
2      $p \leftarrow \emptyset$
3      foreach $n \in operand(i)$ do
4          $p_n \leftarrow$ ConvertToPath(n)
5          $p \leftarrow$ TryMergePath($p$, $p_n$)
6          if $p = \emptyset \vee p_n = \emptyset$ then
7              return $\emptyset$
8      return p
9  Function TryMergePath($p_1$, $p_2$):
10     $p \leftarrow \emptyset$
11     $i_1,\ i_2 \leftarrow 0, 0$
12     while $i_1 < |p_1| \wedge i_2 < |p_2|$ do
13         if $CFGReachable(p_1[i_1],\ p_2[i_2])$ then
14             $p.append(p_1[i_1])$
15             $i_1 \leftarrow i_1 + 1$
16         else if $CFGReachable(p_2[i_2],\ p_1[i_1])$ then
17             $p.append(p_2[i_2])$
18             $i_2 \leftarrow i_2 + 1$
19         else
20             return $\emptyset$
21     return $p$
```

the return value of its callee v_{ret}. The value-set size of other instructions is the product of their operands.

Example 1. *Consider the instruction $I : a = b \ll 2$ in Figure 2, the value-set size of it is two because the size of b at I is two and the size of constant 2 is one. As a validation, we can observe that there are two distinct symbolic formulas for this instruction, including $a = (arg_2 + 4) \ll 2$ and $a = arg_2 \ll 2$.*

2) Prioritized Path Sampling: Next, we perform a prioritized path sampling guided by the value-set size of instructions. Essentially, value-set size indicates the number of symbolic formulas that can be derived from an instruction, estimating the number of distinct runtime values that can be revealed through forced execution. Consequently, our path sampling can be seen as prioritizing paths to reveal each runtime value.

The principle of prioritized sampling consists of two main points. First, we should sample the symbolic formulas of instructions with the smallest value-set sizes first, as instructions with fewer runtime values are more likely to match. Second, we should stop sampling the symbolic formulas of a given instruction when the overlap probability is sufficiently high, since further sampling will yield diminishing returns. This overlap probability is the likelihood of overlap when sampling M items from a set of N elements twice with replacement, and is given by $1 - \frac{\binom{N-M}{M}}{\binom{N}{M}}$. We set the threshold for this probability at $1 - 0.1^4$, which means that the total number of samples taken for an instruction with a value-set size of N is the minimum M such that $\frac{\binom{N-M}{M}}{\binom{N}{M}} \leq 0.1^4$, approximating to the square of N.

The process of sampling formulas for an instruction is similar to calculating its value-set size through data-dependency graph traversal, as previously described. The key difference is that we will record each symbolic formula during graph traversal rather than merely calculating a number.

For each sampled formula, we need to find a block sequence with a valid control-flow topological order that covers the instructions involved in the formula, representing an execution order that can reveal its runtime value. The process is detailed in Alg 1. For each operand of a sampled formula s, the function `ConvertToPath` is invoked recursively to obtain the subpath p_n. Then `TryMergePath` is invoked to merge the subpath p_n with the path p to return (Line 3-7). Path merging involves finding a topological order of the nodes in both paths. This is done by iterating over each node in both paths and checking if a valid topological order exists between them. If a valid order exists, the node with the higher topological order is appended to the merged path (Lines 12-18). Otherwise, it means that a valid CFG path does not exist, and an empty path is returned directly (Lines 19-20). During path merging, we ensure that two consecutive store-load or value-phi nodes in the original paths remain consecutive in the merged path, thereby preserving value dependencies and preventing them from being disrupted by merging with other paths.

Example 2. *Consider the instruction $K : a = b \ll 2$ in Figure 2a, which has a value-set size of 5. As the minimum M for $\frac{\binom{5-M}{M}}{\binom{5}{M}} \leq 0.1^4$ is 3, we sample three symbolic formulas from it. After converting them into block sequences, we obtain three paths $A \to I \to K$, $A \to D \to I \to K$, and $A \to C \to F \to H \to J \to K$, as shown in Figure 2b.*

The time complexity of path sampling is directly determined by the number of paths allowed for sampling each function. In our experiment, we set such a limit to 2,000.

3) Greedy Path Merging: Finally, we merge paths sampled in the previous step to reduce the total number of paths to execute. Two paths can be merged if there is a valid topological sorting of nodes on both paths. Similarly, a group of paths can be merged if every pair of paths in the group can be merged.

If we treat each path as a vertex and an edge to represent whether two paths can be merged, then finding the minimum number of merged paths can be reduced to finding a minimum clique cover in this graph, which is known to be NP-hard [52]. Consequently, we use a greedy approach to efficiently merge as many paths as possible.

The greedy algorithm is presented in Alg 2. Given a set of paths P_{in} to merge, we initialize $P_{remained}$ as P_{in}, which represents the remaining paths to merge (Line 3). In each iteration, we remove one path p from $P_{remained}$, and attempt to merge p with each remaining unmerged path p_x in $P_{remained}$ (Lines 4-7). If p_x can be merged with p, it is also removed from the $P_{remained}$, and p is updated as the merged path (Lines 8-10). At the end of each iteration, the merged path p is added to the merged path set P_{merged} (Line 11).

Algorithm 2: Greedy Path Merging

```
Input: A set of path $P_{in}$ to be merged
Output: A set of merged path $P_{merged}$
Function GreedyMerging(s):
    $P_{merged} \leftarrow \emptyset$
    $P_{remained} \leftarrow P_{in}$
    while $P_{remained} \neq \emptyset$ do
        $p \leftarrow P_{remained}.pop()$
        for $p_x \in P_{remained}$ do
            $p_{merge} \leftarrow$ TryMergePath(p, $p_x$)
            if $p_{merge} \neq \emptyset$ then
                $p \leftarrow p_{merge}$
                $P_{remained}.erase(p_x)$
        $P_{merged}.insert(p)$
    return $P_{merged}$
```

Algorithm 3: Alignment Workflow

```
Input: Reference binary $P_{ref}$
Input: Target binary $P_{tgt}$
Input: Forced execution values $V$ extracted on two binaries
Output: Instruction alignment $M_{inst}$
Output: Basic-block alignment $M_{block}$
Function Workflow(s):
    $M_{inst} \leftarrow \emptyset$
    $M_{block} \leftarrow \emptyset$
    $M_{score} \leftarrow$ GetPairwiseScore($P_{ref}$, $P_{tgt}$, $V$)
    while $HasSinglyTopMatch(M_{score})$ do
        $(f_{ref}, f_{tgt}) \leftarrow$ PopTopMatch($M_{score}$)
        $(m_{inst}, m_{block}) \leftarrow$ GetAlignment($f_{ref}, f_{tgt}, V$)
        $M_{inst} \leftarrow M_{inst} \cup m_{inst}$
        $M_{block} \leftarrow M_{block} \cup m_{block}$
    return $M_{inst}, M_{block}$
```

The merged path set is returned when there are no paths to merge (Line 12).

The greedy approach has worst-case time complexity $O(n^2)$ and best-case time complexity $O(n)$, depending on the proportion of paths that can be merged. In our experiments, we found that most pairs of paths can be merged, making the greedy approach quite efficient in practice. Finally, each merged path is post-processed to include function entry and exit blocks, with sub-paths added between consecutive blocks if they are not directly connected in the CFG.

Example 3. *Consider the five paths sampled on the reference binary as shown in Figure 2b. After merging, we obtain three paths $A \rightarrow B \rightarrow D \rightarrow H \rightarrow I \rightarrow K$, $A \rightarrow C \rightarrow F \rightarrow H \rightarrow I \rightarrow K$, and $A \rightarrow C \rightarrow F \rightarrow H \rightarrow J \rightarrow K$ to be executed.*

B. Forced Execution

The forced execution process follows the standards established in previous work [40], [41], where function parameters are initialized to a fixed set of values. Interpretation is then performed on each instruction in the order of the paths to update the memory state and IR values. Here, we provide a few necessary details on how some important cases are handled.

- **Memory Operation.** Addresses extracted during execution are incomparable due to differences in the address layouts of binaries. To address this issue, we normalize addresses within different memory regions to predefined magic values. Additionally, when a loaded memory is uninitialized, a magic value is returned to indicate the uninitialized state. If a load address belongs to external memory, typically represented by an invalid address (such as a dereference on an argument), a hash value is calculated based on the dereferenced address to represent the value loaded from external memory. This way, if the same external memory address is dereferenced in both binaries, we can ensure that the loaded value remains consistent.
- **Function Calling.** A special case of function calling occurs when a call instruction is not considered an inlined call. In this situation, the execution will still call into the callee function. However, since the execution order within a non-lined called function is not specified, the execution will switch to normal mode, where the direction of a branch instruction is decided by the current execution state.

VI. Instruction Alignment

With the runtime values collected from forced execution, the overall alignment workflow is illustrated in Alg 3. First, pairwise function similarity scores are calculated based on the collected values (Line 4). In each iteration, we retrieve the highest-scoring function pair, and align the instructions and basic blocks of the base functions and their inline callees across the two binaries (Lines 5-7). The alignment results for each function pair are merged into a global map (Lines 8-9), which ultimately serves as the output of BARRACUDA (Line 10). Details of each step are provided in the following subsections.

A. Initial Instruction Alignment

Instruction alignment is a two-stage process. At the first stage, an initial alignment is obtained based on the overlap relationship between instruction semantics.

Definition 2. *(Initial Alignment) We consider instructions I and I' are of initial alignment if $Semantic(I) \cap Semantic(I') \neq \emptyset$ satisfied, denoting as $I \sim I'$*

The above definition indicates that two instructions could align if their semantics overlap. Based on the definition of *Initial Alignment*, we further define *Initial Alignment Targets*.

Definition 3. *(Initial Alignment Targets) For $I_{ref} \in F_{ref}$, the Initial Alignment Targets of I_{ref} are $\{I' \mid I' \sim I, \forall I \in F_{tgt}\}$, denoted by $L(I)$. For $I_{tgt} \in F_{tgt}$, $L(I_{tgt})$ is defined similarly.*

In a word, the initial alignment targets of one instruction are the set of its initial alignments in the counterpart binary.

Example 4. *As Figure 2c shows, for instruction $C : a = a \ll 2$ and instruction $I' : a = b \ll 2$, we have $C \sim I'$, since $Semantic(C) \cap Semantic(I') = \{\ll, 12, 3, 2\}$. Similarly, we have $C \sim C'$. As a result, $L(C) = \{I', C'\}$*

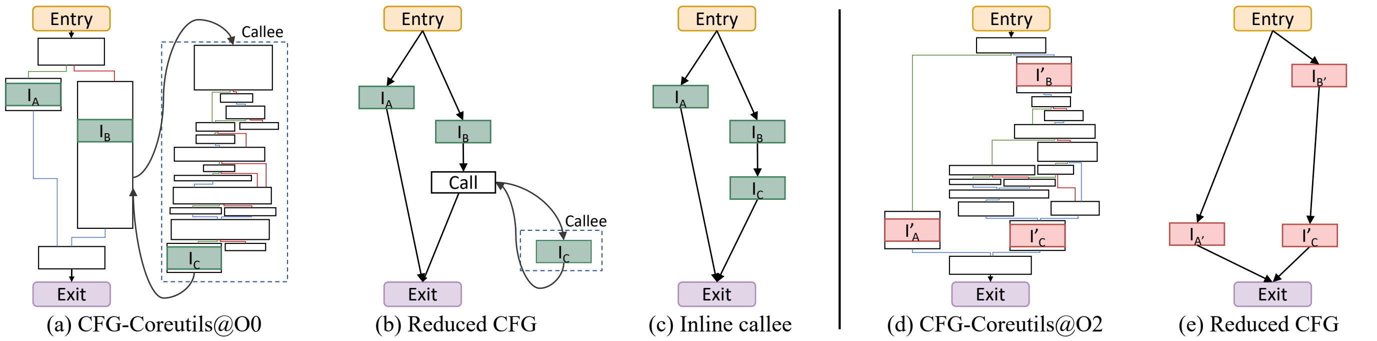


Fig. 6: This figure shows the reduced-CFG construction on *usage_wc* function in Coreutils compiled with O0(left) and O2(right).

Next, we define the *Maximal Alignment Group*.

Definition 4. *(Maximal Alignment Group) We say two sets of instructions S and S' form a Maximal Alignment Group if $\forall I \in S,\ \forall I' \in S',\ S' \subseteq L(I) \land S \subseteq L(I'),\ |S| = |S'|$, and $(S,\ S')$ cannot be extended by adding more instructions while preserving these properties, denoting as $S \approx^{max} S'$.*

Conceptually, if we model the initial alignment relation between instructions as a bipartite graph, then $S \approx^{max} S'$ indicates that S and S' form a *maximal balanced biclique*. Such graph structure is known to be a strong indicator of matching relationships and has been widely applied in domains like bioinformatics [42] and recommendation systems [53]. In our case, $S \approx^{max} S'$ provides a strong hint for the alignment relationship between instructions in two sets.

Example 5. *In Figure 2d, the sets $S : \{C,\ I\}$ and $S' : \{C',\ I'\}$ have the relation $S \approx^{max} S'$, thus there could be a one-to-one alignment between instructions involved.*

The maximal balanced biclique problem (MBB) is known to be NP-hard. However, in our case, the graph established by the initial instruction alignment relationship for a function pair is small and quite sparse, as only a few instruction pairs with the same opcode can have overlapping runtime values between two functions. As a result, we can apply the state-of-the-art MBB algorithm [54] to enumerate maximal alignment groups. Then, if a maximal alignment group contains only a single instruction pair, the instruction pair is directly marked as aligned. For a maximal alignment group containing multiple instructions, we next apply reduced-CFG isomorphism to determine the one-to-one alignment relationships.

B. Reduced-CFG Isomorphism

Aligned instructions should occupy similar positions in their respective CFGs, which is a problem typically solved by graph isomorphism [8], [9]. To ensure scalability and mitigate compiler optimization effects, we propose performing isomorphism on reduced CFGs instead of the original CFGs.

Definition 5. *(Reduced CFG) Given a set of instructions S, its Reduced CFG is the subgraph containing only instructions $I \in S$, while preserving the control-flow ordering: if $I_a \preceq I_b$ in the original CFG, then $I_a \preceq I_b$ in the reduced CFG.*

Reduced-CFG Construction. Given an instruction set S, the construction of a reduced CFG is a two-step process. First, for each basic block in the base function and inlined callees, we delete those basic blocks that do not contain any instruction in S, and the predecessors and successors of the deleted blocks are connected together. For a block containing more than one instruction in set S, the block is split into multiple blocks, where each denotes a specific instruction. If a block contains an inlined callee with instructions in set S, a virtual call node is added and connected to the entry and exit of the callee function. For example, Figure 6a shows the CFG of the $usage\_wc$ function and one of its inlined callees in Coreutils compiled with O0, with a set of instructions $S : \{I_A,\ I_B,\ I_C\}$ marked. The reduced CFG for S is shown in Figure 6b, where all other nodes are removed, and one virtual call node is added.

In the second step, each virtual call is handled by cloning the nodes inside callees into the caller function and replacing the virtual call node. The original callee function is removed when all its virtual call nodes have been processed. Figure 6c shows the final reduced CFG after inlining nodes from the callee. Since we only have one virtual call node, only one instance of the callee nodes is inlined into the base function.

Reduced-CFG Isomorphism. Isomorphism proceeds by depth-first search starting from the entry nodes of two reduced CFGs, recursively checking whether there is a matching between child nodes such that each pair of child nodes is also isomorphic. To handle cases where a node can match multiple nodes due to multiple possible isomorphisms, we enumerate all isomorphism matchings and mark only nodes with exactly one match as final instruction alignments. The isomorphism process has exponential complexity, but in practice, reduced CFGs are often very small because it is difficult to find a large maximal alignment group. Nevertheless, to handle rare cases where the reduced CFG is too large and isomorphism computation takes too long, we set the maximum graph size for isomorphism to 20 in our experiments.

Figure 6d shows the CFG of the $usage\_wc$ function compiled with O2, with a set of instructions $S' : \{I'_A,\ I'_B,\ I'_C\}$ marked. Compared with Figure 6a, we can see significant differences in the syntactic structure between the two CFGs. Specifically, the callee function in Figure 6a is inlined into the base function, and several optimizations are applied to the inlined callee, resulting in some blocks being duplicated or merged. The sets S' and S form a maximal alignment group. However, due to differences in syntactic structure and the large size of the two CFGs, directly applying isomorphism to the original CFGs could be expensive and unlikely to match these instructions. By contrast, when we apply isomorphism on the two reduced CFGs shown in Figure 6c and Figure 6e, a one-to-one instruction alignment can be easily found.

C. Basic-Block Matching

Based on instruction alignment, basic-block matching is also derived and used by downstream diffing tools. We consider two basic blocks B and B' as matched if (1) at least one instruction in B aligns to one instruction in B', and (2) instructions in B and B' do not align to instructions in other basic blocks.

VII. Implementation

Barracuda is built on the LLVM framework [43] and relies on the RetDec decompiler [55], [56] to lift binaries of different architectures into LLVM IR. The main components of Barracuda, which include path sampling, forced execution, and final instruction alignment, comprise about 18,000 lines of C/C++ code. Path sampling is implemented as a series of handlers for different kinds of LLVM instructions, performing graph traversal over the LLVM IR to calculate value-set sizes and sample paths to execute. Similarly, forced execution is implemented as an emulator based on LLVM IR, interpreting the execution semantics of different instruction kinds while updating the abstract state of IR values and abstract memory.

The handling of *load* instructions for value-set size analysis and path sampling requires a data dependency graph to track which IR values can be loaded from memory. We implement the data-dependency following the VLLPA algorithm [57], which is a bottom-up summary-based analysis with context and flow sensitivity. For our purposes, we set the depth of summary inlining to three, as we observe that the maximum inline call depth typically does not exceed this value.

VIII. Evaluation

This section presents the evaluation results of Barracuda by investigating the following research questions:

- **RQ1:** How effective is Barracuda in improving the performance of existing semantic-aware binary diffing?
- **RQ2:** How effectively can Barracuda identify instruction alignment compared to existing work?
- **RQ3:** Is Barracuda efficient enough to be used for enhancing existing binary diffing tools?

A. Experimental Setup

We evaluated Barracuda on a server equipped with four 16-core Intel(R) Xeon Gold 6226R CPU@2.90GHz, four NVIDIA GeForce RTX 3080 GPUs, and 256 GB of RAM. Note that GPUs are used for the execution of other baseline binary diffing tools, and Barracuda does not require GPUs.

Dataset. In our evaluation, we choose the same dataset used by SigmaDiff [10], including Coreutils of three versions (v5.93, v6.4, v8.1), Diffutils of three versions (v2.8, v3.4, v3.6), Findutils of three versions (v4.2.33, v4.4.1, v4.6.0), GMP of three versions (v6.0.0, v6.1.1, v6.2.1), and Putty of three versions (v0.75, v0.76, v0.77). These binaries are compiled with GCC v5.4 and Clang v3.8.0 at four optimization levels (O0, O1, O2, O3). All experiments were conducted on stripped binaries, and debug info is extracted separately using addr2line [58] to serve as ground truth.

Baselines. We consider SigmaDiff [10] and DeepBinDiff [33] as the baseline binary diffing tools to be enhanced by Barracuda. They represent the state-of-the-art semantic-aware binary diffing techniques. We made minimal adaptations to their original official implementation to import the instruction alignment results generated by Barracuda.

- SigmaDiff [10] is a pseudocode-level binary diffing technique. It decompiles binaries into pseudocode tokens using Ghidra (v9.2.2), then relies on DGMC model [36] to perform semi-supervised learning to match pseudocode tokens, where pre-matched tokens are regarded as training nodes. We adapted SigmaDiff's training nodes selection module by also marking two pseudocode tokens as pre-matched nodes if: 1) their corresponding binary instructions are aligned instructions identified by Barracuda, and 2) their opcodes are of the same type.
- DeepBinDiff [33] is designed for basic-block level binary diffing. It utilizes the TADW algorithm [59] to generate semantic-aware basic-block embeddings from a merged CFG of two binaries, then performs k-hop greedy matching to match basic blocks. During its graph merging process, pairs of nodes that are likely to match (e.g., those referring to the same string literal or library function call) are connected by a virtual node, making them easier to match during greedy matching. We adapted DeepBinDiff's graph-merging module to connect two nodes if their corresponding basic blocks are marked as matched by Barracuda.

Evaluation Metrics. We followed the metrics used by SigmaDiff and used the scripts provided in its public GitHub repository to calculate the precision, recall, and F1 score of binary diffing. Specifically, all calculations are based on pseudocode tokens generated by Ghidra [60] with the following formulas:

$$Precision(P) = \frac{C}{M}, \quad Recall(R) = \frac{C}{T}, \quad F_1 = \frac{2PR}{P+R}$$

where C is the number of correct token matches detected by a tool, M is the total number of token matches detected by a tool (including both correct and incorrect matches), and T is the total number of token matches specified in ground truth.

Two tokens are considered correctly matched if their corresponding source-code lines are matched. Since DeepBinDiff outputs basic-block matching pairs instead of pseudocode matches, we follow SigmaDiff's evaluation strategy by marking all pseudocode tokens within the matched basic blocks identified by DeepBinDiff as matched. To evaluate the diffing

TABLE I: Cross-version Pseudocode-Level Diffing Result. (Si: SigmaDiff, De: DeepBinDiff, Ba: Barracuda)

Project	Version	Precision				Recall				F1			
		Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba
Coreutils	v5.93 - v8.1	74.4	**77.4**	72.6	**77.0**	67.7	**71.5**	27.9	**49.0**	70.9	**74.3**	39.6	**59.6**
	v6.4 - v8.1	74.8	**77.3**	75.3	**79.2**	68.5	**71.6**	32.6	**53.4**	71.5	**74.3**	44.4	**63.4**
	Average	74.6	**77.4**	74.0	**78.1**	68.1	**71.5**	30.3	**51.2**	71.2	**74.3**	42.0	**61.5**
Diffutils	v2.8 - v3.6	78.6	**82.2**	**88.5**	87.7	68.9	**73.6**	35.2	**54.5**	73.4	**77.6**	49.9	**67.2**
	v3.4 - v3.6	94.3	**94.9**	94.6	**97.6**	92.3	**93.2**	64.3	**82.8**	93.3	**94.0**	76.3	**89.5**
	Average	86.5	**88.5**	91.5	**92.6**	80.6	**83.4**	49.8	**68.7**	83.4	**85.8**	63.1	**78.3**
Findutils	v4.233 - v4.6	76.2	**77.8**	73.1	**80.7**	69.6	**71.5**	29.7	**48.2**	72.7	**74.5**	42.0	**60.1**
	v4.41 - v4.6	85.0	**85.4**	86.0	**90.4**	80.0	**80.9**	39.0	**58.5**	82.4	**83.1**	53.2	**70.9**
	Average	80.6	**81.6**	79.5	**85.5**	74.8	**76.2**	34.3	**53.3**	77.6	**78.8**	47.6	**65.5**
Gmp	v6.0.0 - v6.2.1	84.6	**87.6**	N/A	N/A	80.0	**83.6**	N/A	N/A	82.2	**85.6**	N/A	N/A
	v6.1.1 - v6.2.1	87.8	**90.1**	N/A	N/A	84.2	**87.3**	N/A	N/A	86.0	**88.7**	N/A	N/A
	Average	86.2	**88.9**	N/A	N/A	82.1	**85.4**	N/A	N/A	84.1	**87.1**	N/A	N/A
Putty	v0.75 - v0.77	67.7	**68.8**	N/A	N/A	64.6	**65.5**	N/A	N/A	66.1	**67.1**	N/A	N/A
	v0.76 - v0.77	68.1	**69.3**	N/A	N/A	65.4	**65.9**	N/A	N/A	66.7	**67.5**	N/A	N/A
	Average	67.9	**69.1**	N/A	N/A	65.0	**65.7**	N/A	N/A	66.4	**67.3**	N/A	N/A
Average		79.2	**81.1**	81.7	**85.4**	74.1	**76.4**	38.1	**57.7**	76.5	**78.7**	50.9	**68.4**
Standard Deviation		±0.04	±0.12	±0.27	±0.21	±0.09	±0.12	±0.24	±0.65	±0.03	±0.12	±0.23	±0.51

on binaries of different versions, we follow existing work and use the Myers algorithm [61] to identify matched lines in the source code. The matched source lines, along with the DWARF debug symbols, are used for the evaluation computation.

Barracuda has parallelism support in its path-sampling and forced-execution stages, while DeepBinDiff and SigmaDiff perform learning in parallel on CPUs and GPUs, respectively. In our experiment, we set CPU parallelism to 16 threads and ran four parallel instances simultaneously on our 64-core, 4-GPU server. All experiments were conducted three times, with averages and standard deviations reported.

B. Effectiveness on Binary Diffing Enhancement

We conducted five sets of evaluations: cross-version, cross-optimization, cross-compiler, cross-architecture, and cross-obfuscation diffing. In cross-obfuscation diffing, binaries were compiled and obfuscated with OLLVM 4.0. Binaries in experiments other than cross-optimization diffing were compiled by O2, while binaries in experiments other than cross-architecture diffing are x86-64 binaries. Except for cross-compiler and cross-obfuscation diffing, all binaries were compiled by GCC v5.4. We set a 24-hour timeout limit for diffing each binary pair. If a tool cannot complete the analysis within the time limit, we mark the corresponding slots in tables as N/A.

Cross-Version Diffing. In the cross-version diffing experiment, we compare the lowest version of each binary in the dataset against each of its higher versions. Table I presents the detailed precision, recall, and F1 score in percentage, where the column `Si` denotes SigmaDiff, `De` denotes DeepBinDiff, and `Si + Ba` and `De + Ba` denote the ones enhanced by importing the anchor points detected by Barracuda.

The data shows that Barracuda consistently improves the F1 scores of both diffing tools across all benchmarks. On average, with Barracuda instruction alignment imported as training nodes, the F1 score of SigmaDiff increases by 2.2% in percentage points. Even on a dataset where the original performance is high, such as the 93.3% F1 score on *Diffutils v3.4 - v3.6*, Barracuda still provides further improvements. The evaluation of DeepBinDiff shows a larger improvement, with the average F1 score increasing from 50.9% to 68.4%. This greater improvement can be attributed to its strong reliance on high-precision anchor points, as its k-hop greedy algorithm starts matching from pairs of pre-matched nodes.

Cross-Optimization Diffing. In this experiment, we compare each O0 binary in the dataset with the one compiled with O1, O2, and O3 optimizations, respectively.

As shown in Table II, with Barracuda, the average F1 scores of SigmaDiff and DeepBinDiff increased by 2.9% and 27.6% in percentage points, respectively. A closer examination of the data reveals that as the gap between optimization levels increases, the improvements from Barracuda become more significant, indicating that precise anchor points are crucial for effective binary diffing in more challenging diffing scenarios. The enhancements observed demonstrate that Barracuda is robust against compiler optimization.

Cross-Compiler Diffing. In this experiment, we compare binaries compiled from GCC v5.4 and Clang v3.8.0.

Table III presents the detailed diffing results. With the fine-grained alignment generated by Barracuda, the precision and recall of both SigmaDiff and DeepBinDiff improved in all cases, resulting in percentage point increases of 4.1% and 42.7% in average F1 score, respectively. Despite that Barracuda is based on LLVM IR, its effectiveness is not affected by the compiler used to compile the binaries. To some extent, using different compilers is similar to compiling with different optimizations. Thus, Barracuda can still generate high-quality instruction alignment to assist with binary diffing.

Cross-Architecture Diffing. In this experiment, we compare binaries compiled for ARM and x86-64 architectures. Since DeepBinDiff does not support ARM binaries, it was excluded from this experiment. As shown in Table IV, Barracuda can significantly improve both the precision and recall of SigmaDiff in all cases, increasing the average F1 score from 53.3% to 56.7%. These promising results further demonstrate the versatility of Barracuda in enhancing

TABLE II: Cross-optimization Pseudocode-level Diffing Result. (Si: SIGMADIFF, De: DEEPBINDIFF, Ba: BARRACUDA)

Project	Version	Precision				Recall				F1			
		Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba
Coreutils	v8.1 O0 - O3	50.1	**54.8**	45.0	**80.0**	35.3	**39.6**	0.8	**23.6**	41.4	**45.9**	1.6	**36.4**
	v8.1 O1 - O3	72.8	**75.2**	64.2	**81.1**	61.5	**64.1**	12.6	**40.7**	66.6	**69.2**	20.6	**54.1**
	v8.1 O2 - O3	**86.3**	86.2	86.1	**91.0**	80.0	**80.2**	51.8	**70.8**	83.0	83.0	64.5	**79.5**
	Average	69.7	**72.1**	65.1	**84.0**	58.9	**61.3**	21.8	**45.0**	63.7	**66.1**	28.9	**56.6**
Diffutils	v3.6 O0 - O3	49.9	**53.7**	50.6	**80.5**	33.7	**37.2**	1.3	**25.0**	40.2	**44.0**	2.6	**38.0**
	v3.6 O1 - O3	73.2	**75.9**	67.2	**81.5**	61.0	**63.4**	14.4	**39.6**	66.5	**69.1**	23.5	**53.3**
	v3.6 O2 - O3	91.5	**91.9**	91.2	**94.4**	87.3	**88.0**	46.9	**70.8**	89.3	**89.9**	61.7	**80.9**
	Average	71.5	**73.8**	69.7	**85.5**	60.6	**62.9**	20.9	**45.1**	65.3	**67.7**	29.2	**57.4**
Findutils	v4.6 O0 - O3	51.7	**56.1**	32.9	**78.6**	35.9	**39.8**	1.0	**24.8**	42.3	**46.6**	2.0	**37.7**
	v4.6 O1 - O3	72.0	**74.6**	66.4	**78.7**	60.3	**62.6**	14.2	**39.1**	65.6	**68.1**	23.3	**52.2**
	v4.6 O2 - O3	88.2	**89.6**	87.5	**91.7**	81.8	**83.2**	49.3	**69.8**	84.9	**86.3**	63.0	**79.2**
	Average	70.6	**73.4**	62.2	**83.0**	59.3	**61.9**	21.5	**44.6**	64.3	**67.0**	29.5	**56.4**
Gmp	v6.2.1 O0 - O3	59.1	**63.1**	N/A	N/A	45.1	**50.0**	N/A	N/A	51.1	**55.8**	N/A	N/A
	v6.2.1 O1 - O3	75.7	**80.1**	N/A	N/A	67.3	**72.1**	N/A	N/A	71.2	**75.9**	N/A	N/A
	v6.2.1 O2 - O3	90.7	**91.2**	N/A	N/A	87.9	**88.2**	N/A	N/A	89.3	**89.7**	N/A	N/A
	Average	75.2	**78.1**	N/A	N/A	66.8	**70.1**	N/A	N/A	70.6	**73.8**	N/A	N/A
Putty	v0.77 O0 - O3	44.3	**50.9**	N/A	N/A	29.0	**33.8**	N/A	N/A	35.0	**40.6**	N/A	N/A
	v0.77 O1 - O3	67.1	**70.7**	N/A	N/A	51.1	**53.8**	N/A	N/A	58.0	**61.1**	N/A	N/A
	v0.77 O2 - O3	78.2	**80.7**	N/A	N/A	63.6	**65.8**	N/A	N/A	70.1	**72.5**	N/A	N/A
	Average	63.2	**67.4**	N/A	N/A	47.9	**51.1**	N/A	N/A	54.4	**58.0**	N/A	N/A
Average		70.1	**73.0**	65.7	**84.2**	58.7	**61.5**	21.4	**44.9**	63.6	**66.5**	29.2	**56.8**
Standard Deviation		±0.09	±0.10	±0.29	±0.31	±0.06	±0.03	±0.06	±0.06	±0.08	±0.04	±0.03	±0.12

TABLE III: Cross-compiler Pseudocode Diffing. (Clang vs. GCC). (Si: SIGMADIFF, De: DEEPBINDIFF, Ba: BARRACUDA)

Project	Precision				Recall				F1			
	Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba
Coreutils v8.1	71.8	**77.1**	76.0	**79.5**	60.1	**64.8**	3.6	**33.5**	65.4	**70.4**	6.9	**47.0**
Diffutils v3.6	75.7	**79.5**	56.6	**76.7**	65.3	**69.7**	2.6	**36.6**	70.1	**74.3**	4.9	**49.5**
Findutils v4.6	76.5	**78.7**	61.6	**77.9**	65.1	**67.6**	4.0	**37.9**	70.4	**72.7**	7.5	**50.9**
Gmp v6.2.1	57.0	**61.9**	N/A	N/A	38.5	**43.3**	N/A	N/A	45.9	**50.9**	N/A	N/A
Putty v0.76	50.1	**54.6**	N/A	N/A	33.1	**36.5**	N/A	N/A	39.8	**43.7**	N/A	N/A
Average	66.2	**70.4**	64.7	**78.0**	52.4	**56.4**	3.4	**36.0**	58.3	**62.4**	6.4	**49.1**
Standard Deviation	±0.17	±0.31	0.78	0.24	±0.07	±0.30	0.07	0.15	±0.10	±0.31	0.15	0.20

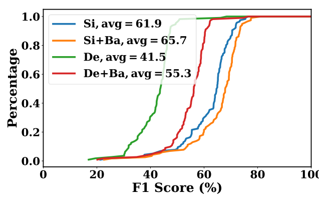
TABLE IV: Cross-architecture Pseudocode-Level Diffing Result (ARM vs. x86-64). (Si: SIGMADIFF, Ba: BARRACUDA)

Project	Precision		Recall		F1	
	Si	Si+Ba	Si	Si+Ba	Si	Si+Ba
Coreutils	67.1	**72.1**	66.3	**71.2**	66.7	**71.6**
Diffutils	70.8	**76.1**	70.6	**75.9**	70.7	**76.0**
Findutils	65.1	**70.0**	64.2	**68.9**	64.6	**69.4**
Gmp	38.1	**39.0**	38.0	**38.9**	38.1	**38.9**
Putty	26.8	**28.3**	25.6	**27.1**	26.2	**27.7**
Average	53.6	**57.1**	52.9	**56.4**	53.3	**56.7**
SD	±0.42	±0.32	±0.36	±0.25	±0.39	±0.28

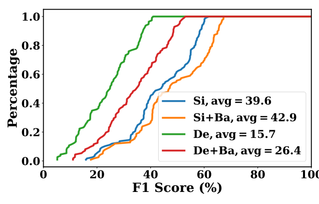
cross-architecture diffing, which is typically more challenging.

Cross-Obfuscation Diffing. In this experiment, we compare each binary compiled by OLLVM-4.0 without obfuscation to those obfuscated by Bogus Control Flow (BCF), Control Flow Flattening (FLA), Instruction Substitution (SUB), and combinations of all three obfuscation methods (ALL). These obfuscations involve semantic-preserving transformations that alter the layout of basic blocks and syntactic representation of instructions.

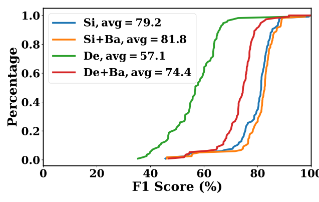
Figure 7 shows the Cumulative Distribution Function (CDF) of the F1 score for each of the four obfuscations, with detailed data provided in Appendix B. On average, BARRACUDA enhances SIGMADIFF with F1 score increases of 3.8%, 3.3%, 2.6%, and 2.3% in percentage points, and enhances DEEPBINDIFF with F1 score increases of 13.8%, 10.7%, 17.3%, and 7.2% for BCF, FLA, SUB, and ALL obfuscation, respectively.



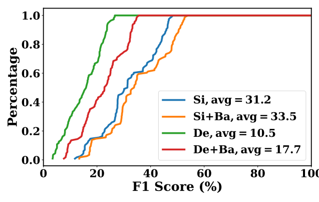
(a) OLLVM-BCF



(b) OLLVM-FLA



(c) OLLVM-SUB



(d) OLLVM-ALL

Fig. 7: Cross-obfuscation Pseudocode Diffing F1-score CDF. (Si: SIGMADIFF, De: DEEPBINDIFF, Ba: BARRACUDA)

These obfuscations can affect the effectiveness of BARRACUDA to some extent. For example, OLLVM-SUB can

TABLE V: Average instruction alignment precision and recall on all the previous experiment settings.

Project	**Literal**		**Literal + Expr**		**Barracuda**		**Barracuda-I**		**Barracuda-V**		**Literal + Expr + Barracuda**	
	Prec	Recl	Prec	Recl	Prec	Recl	Prec	Recl	Prec	Recl	Prec	Recl
Coreutils	91.50	9.80	92.76	30.32	**92.78**	**39.11**	92.66	30.65	92.65	36.89	91.92	45.58
Diffutils	88.10	10.93	**94.61**	29.40	94.33	**40.46**	94.37	30.50	94.15	38.38	93.70	45.58
Findutils	**95.54**	6.68	91.36	28.03	93.08	**36.78**	92.66	28.15	93.10	35.03	91.32	43.22
Gmp	**99.27**	20.76	96.75	**38.51**	94.44	38.03	94.56	35.51	94.45	36.19	94.12	43.30
Putty	**87.70**	10.26	84.45	24.92	85.78	**33.05**	84.71	26.55	85.96	32.10	85.11	36.42
Average	**92.42**	11.68	91.99	30.24	92.08	**37.49**	91.79	30.27	92.06	35.71	91.23	42.82
SD	±0.13	±0.07	±0.04	±0.01	±0.02	±0.02	±0.04	±0.03	±0.05	±0.09	±0.07	±0.27

transform a single arithmetic instruction into multiple instructions that yield the same calculation result, impacting Barracuda's ability to match the obfuscated instruction. However, as a dynamic matching technique, the sampled runtime values are stable against these semantic-preserving obfuscations. Consequently, Barracuda can still identify instruction alignments with higher accuracy than existing anchor-point detection methods and enhance diffing accuracy.

C. *Accuracy of Instruction Alignment*

In this section, we evaluate the accuracy of the instruction alignment generated by Barracuda. For baseline comparison, we selected two early-match strategies used by existing binary diffing. The first strategy matches two instructions if they reference an identical string literal or library function call, which is used by DeepBinDiff and denoted as column Literal in Table V. The second strategy further matches two instructions if symbolic expressions derived from them are syntactically identical, which is the training-node selection technique of SigmaDiff and denoted as column Literal + Expr in Table V. Both techniques are implemented in SigmaDiff, so we used its implementation for comparison.

Since Barracuda and SigmaDiff are based on different IRs, directly calculating their accuracy based on their respective IR instructions could introduce bias. To ensure a fair comparison, we used the binary address of instructions to map the instruction alignment result of Barracuda to the IR of SigmaDiff. In this way, the accuracy of all the instruction alignment techniques can be fairly evaluated based on the IR of SigmaDiff, covering various opcodes such as Load, Store, Cmp, Call, and all kinds of Binary Operators.

Comparison with Existing Techniques. Table V presents the average results collected from all the five experimental settings. Due to differences in binary disassembly and decompilation between SigmaDiff and Barracuda, a small portion of instruction alignments detected by Barracuda cannot be matched to the IR instructions of SigmaDiff. This issue results in some loss in the recall calculation for Barracuda. Despite this disadvantage, the data shows that Barracuda achieves an average recall of 37.49%, significantly higher than the average recalls of 30.24% and 11.68% for the other two strategies, while maintaining a second-highest average precision, only 0.34% slightly slower than Literal.

The last column in Table V presents the merging results, showing an additional 5.33 percentage point improvement in recall compared to using Barracuda alone. This im-

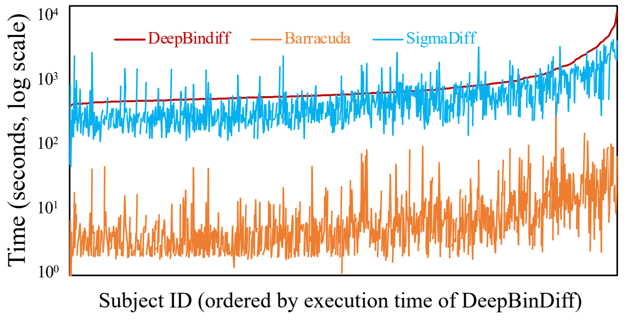


Fig. 8: Execution time of Barracuda and binary diffing.

TABLE VI: Execution Time Summary (in seconds).

	SigmaDiff Enhancement		**DeepBinDiff Enhancement**	
	SigmaDiff	Barracuda	DeepBindiff	Barracuda
Min	40.8	0.2 (+0.49%)	318.5	0.2 (+0.06%)
Max	17026.0	853.6 (+5.01%)	8132.5	443.4 (+5.45%)
Avg	859.3	28.0 (+3.26%)	726.3	12.2 (+1.68%)
SD	±38.35	±1.28	±37.20	±0.54

provement is attributed to the differing core techniques of these strategies. Barracuda is a semantic-aware dynamic alignment technique that can find instruction alignments even when significant syntactic differences arise between binaries. However, due to coverage issues, Barracuda may miss some alignments that other syntactic strategies can detect. The finding indicates the benefit of combining Barracuda with other strategies for significantly better overall performance.

Ablation Study. The column Barracuda-I shows the results when reduced-CFG isomorphism is disabled, meaning that only singly matched pairs are considered as final instruction alignments. The column Barracuda-V shows the results when the prioritized path sampling is disabled. In this case, we utilize the path sampling strategy of existing forced execution tools to cover each basic block once [37], [40].

Compared to Barracuda, the recall for Barracuda-I and Barracuda-V decreases by 7.22% and 1.78% in percentage points, respectively, while both exhibit a slight drop in precision. These decreases in both recall and precision highlight the importance of these components in Barracuda. When Barracuda-V is disabled, we obtain fewer initial alignments because fewer matched instruction pairs in two binaries reveal overlapping instruction semantics. When Barracuda-I is disabled, we cannot determine which instructions match within a maximal alignment group, resulting in fewer instruction pairs being identified as final alignments.

D. *Efficiency*

To assess the additional overhead introduced by applying BARRACUDA to binary diffing enhancement, we compare its execution time with that of the binary diffing tools it improves.

Figure 8 visualizes the execution time of BARRACUDA, SIGMADIFF, and DEEPBINDIFF, derived from diffing Coreutils, Diffutils, and Findutils binaries in cross-version, cross-optimization, cross-compiler, and cross-obfuscation experiments where all three tools are involved. The results show that the execution time for each binary diffing tool is significantly higher than that of BARRACUDA by an order of magnitude.

Table VI presents the comparison results for the minimum, maximum, and average execution time between BARRACUDA and the two downstream binary diffing tools. The comparison with SIGMADIFF is based on all benchmarks across five diffing scenarios, while the comparison with DEEPBINDIFF is based on a subset of the dataset, excluding the Gmp and Putty datasets due to DEEPBINDIFF timeouts, and excluding the cross-architecture diffing experiment because DEEPBINDIFF only supports x86 binaries. On average, BARRACUDA introduces only 3.26% and 1.68% extra runtime overhead for SIGMADIFF and DEEPBINDIFF, respectively, demonstrating that BARRACUDA is efficient enough to enhance the state-of-the-art semantic-aware binary diffing tools. Detailed experimental data on efficiency evaluation can be found in Appendix A.

Execution Time Breakdown. We also further investigate the execution time of different stages in BARRACUDA.

- **Preprocessing**: This stage includes lifting binary into LLVM IR and performing data-dependency analysis. It accounts for 16.5% of the total analysis time.
- **Prioritized Path Sampling**: This stage includes estimating value-set-size and sampling paths to be executed. It accounts for 15.7% of the total analysis time.
- **Forced Execution**: This stage includes executing each sampled path inside emulator to establish instruction and function semantics, occupying 38.0% of analysis time.
- **Instruction Alignment**: This stage leverages collected runtime values to derive initial alignment and performs sub-graph isomorphism to match instructions and basic blocks further, occupying 29.8% of total analysis time.

IX. DISCUSSION

A. *Application Scope and Integration*

BARRACUDA is positioned as a lightweight instruction alignment technique suitable for various binary diffing and comparison scenarios. The experiment demonstrated its advantages in finding fine-grained alignment between binaries with high precision and high efficiency. However, since BARRACUDA does not include a fuzzy-matching stage that relies on machine learning or statistical comparison, it is not suitable for generating comprehensive diffing results on its own. Instead, BARRACUDA is best utilized as a plugin for other binary comparison tools, integrating its high-precision results into their algorithms to enhance overall performance with minimal additional overhead, as discussed in this paper.

Such integration typically requires only a small amount of code adaptation effort. In our experience, integrating BARRACUDA into SIGMADIFF and DEEPBINDIFF requires only a few dozen lines of code. In essence, any binary comparison tool that requires precise, fine-grained alignment could benefit from BARRACUDA. Other potential beneficiaries include patch present analysis [15] and security patch analysis [62], as their techniques also require a precise matching on basic blocks or instructions. However, some techniques are not suitable for integration with BARRACUDA. For example, DIAPHORA [9] treats the pseudocode of binaries as strings and diffs them as a whole, leaving limited space for integrating BARRACUDA.

B. *Limitations*

Function Inlining. BARRACUDA requires information on whether a call site is inlined in the comparison binary to determine the boundary of instruction alignment, which is known as the cross-inlining problem [63]. Current implementation of BARRACUDA applies the rules proposed by asm2vec [22], but is still not accurate enough. Recently, there have been some new explorations [64] on the problem. Integrating the latest of these techniques could also improve BARRACUDA.

One-To-One Matching. Following the limitation of existing work [10], BARRACUDA only supports one-to-one instruction alignment to ensure its high precision. However, because compiler optimizations can duplicate or fuse instructions, some instructions may not be aligned when aggressive optimizations are applied. Handling such cases may require domain knowledge and extensive study of the effects of different compiler optimizations, which can be a challenging problem and a promising future research direction.

Data Dependency Analysis. The prioritized path sampling of BARRACUDA depends on a data dependency graph. Any missing data dependency edges or the presence of incorrect edges can impact the quality of the sampled paths, a challenge that is difficult to avoid due to the difficulty of the problem. Recent advancements in this field [65] have provided new insights, and we envision that deploying such tools could further enhance the performance of BARRACUDA.

Binary Obfuscation and Malware. Although our evaluation demonstrates the effectiveness of BARRACUDA against several OLLVM obfuscations, some obfuscations are difficult to handle. First, since BARRACUDA relies on a static disassembly, it is vulnerable to obfuscations that alter function layout or impede function boundary identification. Examples include inter-procedural obfuscation, such as function splitting [66], and *return-oriented programs* (ROP) [67], which conceal their logic within gadget chains. Second, obfuscations involving self-modification cannot be handled by BARRACUDA as runtime modifications are not modeled. Typical examples include virtualization obfuscations such as VMProtect [68] and Code Virtualizer [69]. To address these limitations, automatic binary deobfuscation techniques [70], [71] could be combined with BARRACUDA. Additionally, since BARRACUDA assumes a memory-safe execution, it is agnostic to post-exploitation behaviors such as remotely injected code and gadgets [72], [73], which are often outside the scope of binary diffing.

Analyze Large Binary. Existing fine-grained binary diffing techniques still cannot scale to very large binaries. While BARRACUDA can improve binary diffing accuracy with only a small relative overhead, it does not enhance scalability, and this limitation remains. Exploring more efficient fine-grained diffing methods could be an important future research topic.

X. Related Work

A. Binary Diffing

Function Similarity Analysis. A majority of binary diffing focuses on computing similarity scores between functions. Traditional approaches rely on the syntactic structure of CFG [1], [8], [19] or partial traces decomposed from CFG [3], [14], [20] for similarity computation. Several methods leverage advances in deep learning to compare similarities based on graph embeddings extracted from CFG [4], [5], [21], [74]–[79] or dependencies between instructions [22], [24], [25], [32], [80]–[84]. To address cross-compilation similarity, re-optimization techniques [23], [85] were proposed. Dynamic analysis has also been widely applied to capture runtime behaviors for similarity comparison [38], [39], [86], [87]. To address poor coverage issues, forced execution [37], [40], [41] is further proposed to execute binaries while overriding intended program logic. However, they all aim for high block coverage instead of path coverage; as a result, they are ineffective when used for fine-grained instruction alignment.

Basic-Block Matching. Theorem provers are commonly utilized to determine the semantic equivalence between basic blocks [6], [26], [27], [30], [31], [34]. Multi-MH [29] and Bingo [88] break binary code into smaller fragments for semantic extraction and comparison. VSIM [28] enhances efficiency with under-constrained symbolic execution values. BinDiff [8] and Diaphora [9] match basic blocks between functions based on CFG isomorphism and matching heuristics, while DEEPBINDIFF [33] further improves these methods with the TADW algorithm [59] and context- and semantic-aware basic block embeddings. BARRACUDA also exploits the isomorphic structure of CFGs for alignment. However, instead of matching the full CFG, we only perform isomorphism on a smaller reduced CFG to match instructions within an alignment group, resulting in greater precision and efficiency.

Instruction Alignment. Recently, instruction-level binary diffing has gained attention. Diaphora [9] performs instruction-level diffing via string-level diffing. SIGMADIFF [10] is the first general static method for instruction-level diffing, defining signatures as symbolic formulas of instructions and leveraging the DGMC graph-matching model to match IR instructions. Many patch presence analysis methods [14]–[16], [89] also perform instruction-level matching to determine whether patch signatures exist in a target binary; however, such alignment typically only operates within a relatively small code scope rather than providing a general approach. DTW [35] and BinSim [34] also conduct instruction alignment through dynamic analysis. However, their approaches focus on aligning instruction traces logged from concrete execution under specific inputs, whereas BARRACUDA aligns instructions using forced execution, eliminating the need for specific input.

B. Program Analysis via Path Sampling

Path sampling is widely applied in various applications, including binary similarity [37], [40], [41], malware analysis [90]–[93], data dependency analysis [94], and fault localization [95]. Different approaches have different focuses on path prioritization. For example, Pem [41] samples equivalent paths via a probability model, BDA [94], [96] samples paths with equal probability, and Empc [97] samples paths as a minimum path cover problem to increase symbolic execution coverage. In our work, we sample paths to favor revealing values more likely to lead to instruction alignment.

XI. Conclusion

This paper proposes BARRACUDA, a high-confidence dynamic instruction alignment. Extensive evaluation demonstrates that BARRACUDA can detect 24.0% more instruction-level matches with a high precision of 92.1%, improving two state-of-the-art binary diffing tools, DEEPBINDIFF and SIGMADIFF, with F1 score percentage-point increases ranging from 12.3% to 42.7% and 2.2% to 4.1%, respectively.

Acknowledgment

We thank reviewers for their valuable comments on this work. This work is supported by research donations from Huawei, TCL, and Tencent. Anshunkang Zhou is the corresponding author.

References

[1] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[2] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *SIGPLAN Not.*, vol. 53, no. 2, p. 392–404, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3177157

[3] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 376–387. [Online]. Available: https://doi.org/10.1145/3395363.3397361

[4] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[5] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 896–899.

[6] 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.

[7] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 756–765.

[8] "Zynamics bindiff," https://www.zynamics.com/bindiff.html, 2025.

[9] "Diaphora," https://github.com/joxeankoret/diaphora, 2025.

[10] L. Gao, Y. Qu, S. Yu, Y. Duan, and H. Yin, "Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing," *Proceedings 2024 Network and Distributed System Security Symposium*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:262144278

[11] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 769–780.

[12] H. Wang, Z. Liu, S. Wang, Y. Wang, Q. Tang, S. Nie, and S. Wu, "Are we there yet? filling the gap between binary similarity analysis and binary software composition analysis," in *2024 IEEE 9th European Symposium on Security and Privacy*, 2024, pp. 506–523.

[13] H. Wang, Z. Liu, Y. Dai, S. Wang, Q. Tang, S. Nie, and S. Wu, "Preserving privacy in software composition analysis: A study of technical solutions and enhancements," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, p. 2329–2341.

[14] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1149–1163. [Online]. Available: https://doi.org/10.1145/3372297.3417240

[15] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 887–902.

[16] Q. Zhan, X. Hu, X. Xia, and S. Li, "React: Ir-level patch presence test for binary," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 381–392.

[17] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu, "sem2vec: Semantics-aware assembly tracelet embedding," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, May 2023.

[18] W. K. Wong, H. Wang, Z. Li, and S. Wang, "BinAug: Enhancing binary similarity analysis with low-cost input repairing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[19] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ser. PPREW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2430553.2430557

[20] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 349–360. [Online]. Available: https://doi.org/10.1145/2594291.2594343

[21] 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*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 363–376. [Online]. Available: https://doi.org/10.1145/3133956.3134018

[22] S. H. H. Ding, B. C. M. 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)*, 2019, pp. 472–489.

[23] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 79–94. [Online]. Available: https://doi.org/10.1145/3062341.3062387

[24] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3236–3251. [Online]. Available: https://doi.org/10.1145/3460120.3484587

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

[26] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 346–359. [Online]. Available: https://doi.org/10.1145/3052973.3052995

[27] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *SIGPLAN Not.*, vol. 51, no. 6, p. 266–280, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908126

[28] H. Wang and Z. Lin, "vSim: Semantics-aware value extraction for efficient binary code similarity analysis," in *Network and Distributed Systems Security (NDSS) Symposium*, 2026.

[29] 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*, 2015, pp. 709–724.

[30] 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*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 238–255. [Online]. Available: https://doi.org/10.1007/978-3-540-88625-9_16

[31] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with inter-procedural control flow," in *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, T. Kwon, M. Lee, and D. Kwon, Eds., vol. 7839. Springer, 2012, pp. 92–109. [Online]. Available: https://doi.org/10.1007/978-3-642-37682-5_8

[32] 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 *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[33] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[34] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 253–270.

[35] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 342–352.

[36] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege, "Deep graph matching consensus," *International Conference on Learning Representations (ICLR)*, 2020.

[37] A. Zhou, Y. Hu, X. Xu, and C. Zhang, "Arcturus: Full coverage binary similarity analysis with reachability-guided emulation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3640337

[38] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 319–330.

[39] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 57–67.

[40] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 303–317.

[41] X. Xu, Z. Xuan, S. Feng, S. Cheng, Y. Ye, Q. Shi, G. Tao, L. Yu, Z. Zhang, and X. Zhang, "Pem: Representing binary program semantics for similarity analysis via a probabilistic execution model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 401–412. [Online]. Available: https://doi.org/10.1145/3611643.3616301

[42] Y. Cheng and G. M. Church, "Biclustering of expression data." in *Ismb*, vol. 8, no. 2000, 2000, pp. 93–103.

[43] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[44] I. Rocco, M. Cimpoi, R. Arandjelović, A. Torii, T. Pajdla, and J. Sivic, "Neighbourhood consensus networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 1658–1669.

[45] J. Bian, W.-Y. Lin, Y. Matsushita, S.-K. Yeung, T.-D. Nguyen, and M.-M. Cheng, "Gms: Grid-based motion statistics for fast, ultra-robust feature correspondence," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2828–2837.

[46] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 833–851.

[47] G. Balakrishnan and T. Reps, "Divine: discovering variables in executables," in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 1–28.

[48] G. Balakrishnan and R. Thomas, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[49] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, Aug. 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[50] S. H. Kim, D. Zeng, C. Sun, and G. Tan, "Binpointer: towards precise, sound, and scalable binary-level pointer analysis," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 169–180.

[51] C. Ye, Y. Cai, A. Zhou, H. Huang, H. Ling, and C. Zhang, "Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 170–187. [Online]. Available: https://doi.org/10.1145/3622781.3674177

[52] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 2009, pp. 219–241.

[53] C. Maier and D. Simovici, "On biclique connectivity in bipartite graphs and recommendation systems," in *Proceedings of the 2021 5th International Conference on Information System and Data Mining*, ser. ICISDM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 151–156. [Online]. Available: https://doi.org/10.1145/3471287.3471302

[54] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient exact algorithms for maximum balanced biclique search in bipartite graphs," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 248–260.

[55] P. M. J. Kˇroustek, "Retdec: An open-source machine-code decompiler," Presented at Pass the SALT 2018, Lille, FR, July 2018.

[56] A. Zhou, C. Ye, H. Huang, Y. Cai, and C. Zhang, "Plankton: Reconciling binary code and debug information," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 912–928. [Online]. Available: https://doi.org/10.1145/3620665.3640382

[57] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05. USA: IEEE Computer Society, 2005, p. 291–302. [Online]. Available: https://doi.org/10.1109/CGO.2005.27

[58] "addr2line," https://sourceware.org/binutils/docs/binutils/addr2line.html, 2025.

[59] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, p. 2111–2117.

[60] N. S. Agency, "Ghidra reverse engineering tool," https://ghidra-sre.org/, 2025.

[61] E. W. Myers, "Ano(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1–4, p. 251–266, Nov. 1986. [Online]. Available: https://doi.org/10.1007/BF01840446

[62] 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)*, 2017, pp. 462–472.

[63] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3561385

[64] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, and T. Liu, "Cross-inlining binary function similarity detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639080

[65] L. Gao and H. Yin, "Bindsa: Efficient, precise binary-level pointer analysis with context-sensitive heap reconstruction," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1190–1211, 2025.

[66] P. Zhang, C. Wu, M. Peng, K. Zeng, D. Yu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 55–67. [Online]. Available: https://doi.org/10.1145/3579990.3580007

[67] G. Poulios, C. Ntantogian, and C. Xenakis, "Ropinjector: Using return oriented programming for polymorphism and antivirus evasion," *Blackhat USA*, 2015.

[68] V. Software, "Vmprotect software protection," http://vmpsoft.com, last reviewed: 11/13/2025.

[69] O. Technologies, "Code virtualizer: Total obfuscation against reverse engineering," http://oreans.com/codevirtualizer.php, last reviewed: 11/13/2025.

[70] K. Lu, D. Zou, W. Wen, and D. Gao, "derop: removing return-oriented programming from malware," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 363–372. [Online]. Available: https://doi.org/10.1145/2076732.2076784

[71] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 674–691.

[72] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: when benign apps become evil," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USA: USENIX Association, 2013, p. 559–572.

[73] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[74] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2253–2265.

[75] H. Wang, P. Ma, Y. Yuan, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Enhancing DNN-based binary code function search with low-cost equivalence checking," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 226–250, 2022.

[76] 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*, ser. ASE'18. New York, NY, USA: ACM, 2018.

[77] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: unlock the power of semantics-oriented graph representation for binary code similarity detection," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[78] J. Wang, C. Zhang, L. Chen, Y. Rong, Y. Wu, H. Wang, W. Tan, Q. Li, and Z. Li, "Improving ml-based binary function similarity detection by assessing and deprioritizing control flow graph features," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[79] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, “Binaryai: Binary software composition analysis via intelligent binary source code matching,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[80] 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.
[81] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, “Codee: A tensor embedding scheme for binary code search,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2021.
[82] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, “Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection,” in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 224–236.
[83] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “Jtrans: Jump-aware transformer for binary code similarity detection,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.
[84] K. He, Y. Hu, X. Li, Y. Song, Y. Zhao, and D. Gu, “Strtune: Data dependence-based code slicing for binary similarity detection with fine-tuned representation,” *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 10 233–10 245, 2024.
[85] J. Jiang, G. Li, M. Yu, G. Li, C. Liu, Z. Lv, B. Lv, and W. Huang, “Similarity of binaries across optimization levels and obfuscation,” in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 295–315.
[86] Y. Hu, Y. Zhang, J. Li, and D. Gu, “Binary code clone detection across architectures and compiling configurations,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 88–98.
[87] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, “Binmatch: A semantics-based hybrid approach on binary code clone analysis,” in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 104–114.
[88] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, “Bingo: Cross-architecture cross-os binary search,” in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 678–689.
[89] Q. Zhan, X. Hu, Z. Li, X. Xia, D. Lo, and S. Li, “Ps3: Precise patch presence test based on semantic symbolic signature,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
[90] Z. Tang, J. Zhai, M. Pan, Y. Aafer, S. Ma, X. Zhang, and J. Zhao, “Dual-force: Understanding webview malware via cross-language forced execution,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 714–725.
[91] J. Wilhelm and T.-c. Chiueh, “A forced sampled execution approach to kernel rootkit identification,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 219–235.
[92] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, “Pmp: Cost-effective forced execution with probabilistic memory pre-planning,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1121–1138.
[93] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, “X-force: Force-executing binary programs for security applications,” in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 829–844.
[94] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, “Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360563
[95] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 272–281.
[96] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, “Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.
[97] S. Yao and D. She, “ Empc: Effective Path Prioritization for Symbolic Execution with Path Cover ,” in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2772–2790. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00190

TABLE VII: Average runtime overhead on each benchmark across different diffing scenarios. (Si: SigmaDiff, De: DeepBinDiff, Ba: Barracuda)

Project	Size	Num	Scenario	Average Execution Time (s)		
				De	Si	Ba
Coreutils	67K	1120	Ver	548.2	298.6	5.2
			Opt	679.4	386.7	6.5
			Comp	548.0	419.9	8.9
			Arch	N/A	622.2	10.6
			Obf	811.1	538.7	19.6
Diffutils	99K	44	Ver	610.5	448.1	7.4
			Opt	750.8	548.1	8.7
			Comp	603.0	545.6	8.4
			Arch	N/A	908.4	10.6
			Obf	996.7	731.3	24.8
Findutils	164K	33	Ver	1954.7	1140.3	10.7
			Opt	1795.1	872.7	13.3
			Comp	1226.4	903.8	13.5
			Arch	N/A	1913.1	17.7
			Obf	2540.8	1240.3	29.9
Gmp	682K	11	Ver	N/A	4604.2	651.1
			Opt	N/A	5278.8	783.7
			Comp	N/A	6881.9	556.5
			Arch	N/A	6012.7	595.7
			Obf	N/A	8120.8	514.4
Putty	792K	44	Ver	N/A	8891.4	247.2
			Opt	N/A	7686.5	375.1
			Comp	N/A	5846.5	216.0
			Arch	N/A	6512.6	243.5
			Obf	N/A	13435.7	339.0

Appendix A
Runtime Overhead Details

Table VII further presents the detailed average runtime overhead of SigmaDiff, DeepBinDiff, and Barracuda across five benchmark projects and all five diffing scenarios. The `Size` column indicates the average size of the stripped binaries involved in the diffing experiments, and the `Num` column represents the number of binary-diffing pairs performed in these experiments. In the `Scenario` column, Ver, Opt, Comp, Arch, and Obf abbreviate Cross-Version, Cross-Optimization, Cross-Compiler, Cross-Architecture, and Cross-Obfuscation diffing scenarios, respectively.

Generally speaking, larger binary pairs require more time for analysis with all three tools. Both SigmaDiff and DeepBinDiff analyze the inter-procedural CFG or DDG of a binary as a whole during their core diffing processes, so their execution time is directly influenced by the number of instructions in the binary. In contrast, Barracuda is more sensitive to the size and number of large functions in binaries, as the path sampling is conducted on a per-function basis. The larger a function, the more paths and longer paths are required for sampling to reveal instruction semantics, resulting in longer sampling time. For instance, Barracuda takes more time on the Gmp benchmark due to the presence of many large functions used for high-precision arithmetic calculations.

Appendix B
Cross-Obfuscation Diffing Details

Table VIII shows the detailed result of the cross-obfuscation diffing on all five benchmarks, including Coreutils v8.1,

TABLE VIII: Cross-obfuscation Pseudocode-level Diffing Result. (Si: SIGMADIFF, De: DEEPBINDIFF, Ba: BARRACUDA)

Project	Version	Precision				Recall				F1			
		Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba	Si	Si+Ba	De	De+Ba
Coreutils	OLLVM-BCF	68.4	**71.7**	72.2	**78.1**	56.6	**59.9**	29.8	42.6	**61.9**	**65.2**	41.9	**54.9**
	OLLVM-FLA	58.1	**63.0**	64.0	**73.9**	35.7	**39.3**	15.1	**22.2**	44.0	**48.2**	24.0	**33.6**
	OLLVM-SUB	83.4	**84.9**	90.0	**94.3**	75.8	**77.4**	41.1	**60.0**	79.4	**80.9**	56.0	**73.1**
	OLLVM-ALL	47.3	**51.7**	41.6	**53.4**	25.4	**28.1**	9.7	**14.6**	32.9	**36.3**	15.4	**22.7**
	Average	64.3	**67.8**	67.0	**74.9**	48.4	**51.2**	23.9	**34.9**	54.6	**57.7**	34.3	**46.0**
Diffutils	OLLVM-BCF	68.6	**71.7**	76.6	**83.6**	57.0	**60.7**	34.3	**48.5**	62.3	**65.7**	47.2	**61.3**
	OLLVM-FLA	43.1	**45.7**	60.3	**74.4**	26.1	**28.4**	5.7	**12.9**	32.5	**35.0**	10.4	**22.0**
	OLLVM-SUB	86.1	**87.5**	94.8	**96.9**	79.3	**81.2**	45.9	**61.5**	82.6	**84.2**	61.8	**75.3**
	OLLVM-ALL	39.3	**41.3**	42.3	**58.7**	19.6	**21.8**	3.9	**8.2**	26.1	**28.5**	7.0	**14.4**
	Average	59.3	**61.6**	68.5	**78.4**	45.5	**48.0**	22.4	**32.8**	50.9	**53.3**	31.6	**43.2**
Findutils	OLLVM-BCF	67.2	**69.6**	71.4	**78.4**	49.3	**52.2**	24.2	**37.7**	56.3	**59.1**	35.4	**49.8**
	OLLVM-FLA	45.4	**49.0**	47.7	**66.2**	25.1	**27.2**	7.5	**14.7**	31.9	**34.6**	12.7	**23.6**
	OLLVM-SUB	88.7	**89.7**	90.2	**95.3**	79.6	**80.8**	38.0	**61.6**	83.8	**84.9**	53.4	**74.8**
	OLLVM-ALL	37.6	**40.9**	34.9	**47.8**	18.7	**20.4**	5.5	**9.9**	24.7	**26.9**	9.2	**16.0**
	Average	59.7	**62.3**	61.0	**71.9**	43.2	**45.2**	18.8	**31.0**	49.2	**51.4**	27.7	**41.1**
Gmp	OLLVM-BCF	81.2	**85.3**	N/A	N/A	72.4	**77.6**	N/A	N/A	76.5	**81.3**	N/A	N/A
	OLLVM-FLA	62.1	**66.0**	N/A	N/A	44.4	**48.2**	N/A	N/A	51.8	**55.7**	N/A	N/A
	OLLVM-SUB	80.4	**85.1**	N/A	N/A	72.6	**78.8**	N/A	N/A	76.3	**81.8**	N/A	N/A
	OLLVM-ALL	58.9	**59.5**	N/A	N/A	38.8	**39.5**	N/A	N/A	46.8	**47.5**	N/A	N/A
	Average	70.7	**74.0**	N/A	N/A	57.1	**61.1**	N/A	N/A	62.9	**66.6**	N/A	N/A
Putty	OLLVM-BCF	62.2	**67.5**	N/A	N/A	45.2	**49.6**	N/A	N/A	52.4	**57.2**	N/A	N/A
	OLLVM-FLA	52.2	**55.7**	N/A	N/A	29.8	**32.4**	N/A	N/A	37.9	**41.0**	N/A	N/A
	OLLVM-SUB	80.9	**84.3**	N/A	N/A	68.3	**71.6**	N/A	N/A	74.1	**77.4**	N/A	N/A
	OLLVM-ALL	41.5	**45.0**	N/A	N/A	18.7	**20.7**	N/A	N/A	25.7	**28.3**	N/A	N/A
	Average	59.2	**63.1**	N/A	N/A	40.5	**43.6**	N/A	N/A	47.5	**51.0**	N/A	N/A
Average		62.6	**65.8**	65.5	**75.1**	46.9	**49.8**	21.7	**32.9**	53.0	**56.0**	31.2	**43.5**
Standard Deviation		±0.21	±0.08	±0.46	±0.19	±0.06	±0.09	±0.22	±0.10	±0.09	±0.07	±0.25	±0.16

Diffutils v3.6, Findutils v4.6, Gmp v6.2.1, and Putty v0.77, corresponding to the Cumulative Distribution Function result shown in Figure 7 in §VIII.B. On average, with BARRACUDA, the average F1 scores of SIGMADIFF and DEEPBINDIFF increase by 3.0% and 12.3% in percentage points, respectively.

APPENDIX C
ARTIFACT APPENDIX

This artifact includes the implementation, experimental scripts, environmental dependencies, and datasets required to reproduce the experiments described in the paper, as well as the original data supporting the presented results.

A. Description & Requirements

1) How to Access: The complete artifact file is available on Zenodo at the following DOI link: https://doi.org/10.5281/zenodo.17885726. The Docker environment with environmental dependency set up can be accessed at https://hub.docker.com/r/cyeaa/barracuda.

2) Hardware Dependencies: The evaluation was conducted on a server equipped with four 16-core Intel(R) Xeon Gold 6226R CPUs at 2.90GHz, four NVIDIA GeForce RTX 3080 GPUs, and 256 GB of RAM.

3) Software Dependencies: The baseline tools SIGMADIFF and DEEPBINDIFF require different Python dependencies, as well as JDK 11 and Ghidra 9.2.2, all of which are included as pre-installed bundles within the Docker container.

4) Benchmarks: The experiments described in the paper include five benchmarks: Coreutils, Diffutils, Findutils, Gmp, and Putty. A complete experiment requires approximately one week of execution on a server equipped with hardware similar to that described in *Hardware Dependencies*. We provide both full-scale experiment scripts for evaluation on all benchmarks and downscaled experiment scripts for the Diffutils benchmark. The full-scale dataset and scripts are available in the archive `AE_full.tar.xz` on Zenodo, while the downscaled dataset and scripts are in the archive `AE.tar.xz` on Zenodo. The original raw data and results can be downloaded via the following link: https://hkustconnect-my.sharepoint.com/:u:/g/personal/cyeaa_connect_ust_hk/IQBtam_45abBQKcXWrTldatsASp-EcYq4sco9d2GZxgC3x4?e=rmt5On.

B. Artifact Installation & Configuration

The home directory inside the Docker container contains the dataset and scripts for the downscaled evaluation, with the folder `diff_exp` containing the entry scripts to start the evaluation. A full-scale evaluation can be performed by downloading and extracting the archive `AE_full.tar.xz` to the home directory. In this case, the `diff_full_exp` folder contains the entry scripts.

To obtain the Docker image:

```
$ docker pull cyeaa/barracuda:2.0
```

To create a Docker container with the Docker image:

```
$ docker run --gpus all -it
cyeaa/barracuda:2.0
```

C. Major Claims

- (C1): BARRACUDA can detect precise instruction alignments as anchor points, effectively enhancing the accuracy of state-of-the-art fine-grained binary diffing across various scenarios. This is demonstrated by experiments (E1) and (E2), with results reported in Tables I, II, III, IV, V, and VIII.

- (C2): BARRACUDA introduces only minimal additional runtime overhead to existing binary diffing tools. This is demonstrated by experiment (E3), with results reported in Figure 7 and §VIII.D.

D. Evaluation

This section describes the process for reproducing the downscaled experiment. The full-scale experiment can be reproduced similarly, with slight manual adjustments recommended to run it in multiple batches simultaneously (based on the server hardware specifications used) to accelerate the process. It took approximately 1 week to complete one round of the full-scale experiment, with 4 batch executions running in parallel in our experiment environment.

1) Experiment (E0): **[10 human-minutes]**: Download the artifact archive `AE.tar.xz` and extract it in the home directory (if you are using Docker, you can skip this step). Then, enter the `diff_exp` folder, where all subsequent experiments will be conducted.

2) Experiment (E1): **[10 human-minutes + 12 compute-hours]**: This experiment performs cross-version, cross-optimization, cross-architecture, cross-compiler, and cross-obfuscation diffing on the Diffutils dataset. Our evaluation metrics include precision, recall, and F1-score. The experiment is conducted using SigmaDiff and DeepBindiff, and the corresponding versions accepting the BARRACUDA instruction alignment results as anchor points.

[Execution]: Run `./run_exp1.sh`. After the experiment finishes, the diffing output generated by the different experiment groups will be placed in their corresponding subfolders.

[Results]: Run `./show_exp1.sh`. This will generate evaluation results in the standard output, including data rows corresponding to Diffutils in Tables I, II, III, IV, and VIII. Slight variations in specific data points are expected, but the comparison trends across different tools should remain largely consistent.

3) Experiment (E2): **[10 human-minutes + 2 compute-hours]**: This experiment evaluates the accuracy of instruction alignments generated by BARRACUDA using two ablation groups and compares them with two anchor point detection methods implemented in SIGMADIFF.

[Execution]: Run `./run_exp2.sh`. After the experiment finishes, the diffing output will be placed in the folder `TrainingNodeEval`.

[Results]: Run `./show_exp2.sh`. This will generate evaluation results in the standard output, including data rows corresponding to Diffutils in Table V. Slight variations in specific data points are expected due to a few random factors in BARRACUDA, such as path sampling, but the comparison trends across different evaluation groups should remain largely consistent.

4) Experiment (E3): **[5 human-minutes]**: This experiment displays the comparison of runtime overhead between BARRACUDA, SIGMADIFF, and DEEPBINDIFF, based on the results obtained from previous executions.

[Results]: Run `./show_exp3.sh`. This will generate evaluation results in the standard output, including data corresponding to Figure 7 and §VIII.D.