ARCTURUS: Full Coverage Binary Similarity Analysis with Reachability-guided Emulation

ANSHUNKANG ZHOU, Hong Kong University of Science and Technology, Hong Kong, China
YIKUN HU, Shanghai Jiao Tong University, Shanghai, China
XIANGZHE XU, Purdue University, West Lafayette, USA
CHARLES ZHANG, Hong Kong University of Science and Technology, Hong Kong, China

Binary code similarity analysis is extremely useful, since it provides rich information about an unknown binary, such as revealing its functionality and identifying reused libraries. Robust binary similarity analysis is challenging, as heavy compiler optimizations can make semantically similar binaries have gigantic syntactic differences. Unfortunately, existing semantic-based methods still suffer from either incomplete coverage or low accuracy.

In this article, we propose ARCTURUS, a new technique that can achieve high code coverage and high accuracy simultaneously by manipulating program execution under the guidance of code reachability. Our key insight is that the compiler must preserve program semantics (e.g., dependences between code fragments) during compilation; therefore, the code reachability, which implies the interdependence between code, is invariant across code transformations. Based on the above insight, our key idea is to leverage the stability of code reachability to manipulate the program execution such that deep code logic can also be covered in a consistent way. Experimental results show that ARCTURUS achieves an average precision of 87.8% with 100% block coverage, outperforming compared methods by 38.4%, on average. ARCTURUS takes only 0.15 second to process one function, on average, indicating that it is efficient for practical use.

CCS Concepts: • **Security and privacy** → **Software reverse engineering;**

Additional Key Words and Phrases: Binary code similarity, emulation, reverse engineering, program analysis

ACM Reference Format:
Anshunkang Zhou, Yikun Hu, Xiangzhe Xu, and Charles Zhang. 2024. ARCTURUS: Full Coverage Binary Similarity Analysis with Reachability-guided Emulation. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 96 (April 2024), 31 pages. https://doi.org/10.1145/3640337

1 INTRODUCTION

Binary similarity analysis answers to what extent two pieces of binary code are similar and determines if they come from the same codebase. It is a fundamental technique in various security applications such as vulnerability discovery [15, 29–31], malware detection [13, 42, 60, 61], patch analysis [45, 96, 99], forensics [56, 67], and component analysis [44, 72, 74, 91, 95]. For example,

Authors' addresses: A. Zhou and C. Zhang, Hong Kong University of Science and Technology, Clear Water Bay, Sai Kung, Hong Kong, China; e-mails: {azhouad, charlesz}@cse.ust.hk; Y. Hu (Corresponding author), Shanghai Jiao Tong University, Shanghai, China; e-mail: yikunh@sjtu.edu.cn; X. Xu, Purdue University; e-mail: xu1415@purdue.edu.

ACM 1049-331X/2024/04-ART96
https://doi.org/10.1145/3640337

a recent study shows that 95% of software applications reuse third-party components [2]; therefore, identifying code fragments that are similar to known codebases allows us to apply existing information to facilitate code understanding.

Considerable research efforts have been dedicated to the binary similarity analysis problem due to its importance. Existing works mainly tackle the problem from two different angles. First, some approaches [3, 20, 22, 33, 86] extract syntactic or structural code features such as **control-flow graphs (CFGs)** and instruction sequences to compare binary code. However, they are not robust enough, because binaries compiled from the same source code could have different syntax and structures due to various code transformations adopted by compilers, such as loop unrolling [23] and common expression elimination [78]. Second, some methods identify similar code fragments by reasoning about their functionalities (semantics), which are the input/output values of procedures [19, 37, 58]. They usually work by executing the target code with generated inputs to collect runtime values, and two pieces of code are considered similar if their runtime values have similarities. They are more robust against code transformations, as code with similar semantics must have similar dynamic behaviors under the same input regardless of synthetic differences [28]. However, due to the inherent coverage issue in the dynamic analysis [36], these methods can hardly distinguish programs that have similar shallow paths [70] (e.g., error handling code) but with different deeply embedded behaviors [48]. Although IMF-Sim [80] leverages advances in fuzzing techniques to increase code coverage, it only covers 31.8% of the instructions.

Several approaches [17, 28, 65] propose to increase code coverage by overriding the intended program logic (branches) during the execution, which is also referred to as *forced execution* [64]. They usually traverse the code in a way that only considers the CFG structure. For example, BLEX [28] executes the target function repeatedly, starting from so far uncovered instructions until every instruction is executed at least once. However, since code optimizations and transformations could radically modify the program CFGs, forced execution-based techniques cannot guarantee consistent runtime behaviors between semantically similar but synthetically different code; thus, the accuracy is still limited [26, 54].

In this article, we propose ARCTURUS, a new technique that can achieve high code coverage and high accuracy simultaneously by manipulating program execution under the guidance of code reachability. Our key insight is that the compiler must preserve program semantics (e.g., dependences between code fragments) during compilation; therefore, the code reachability, which implies the interdependence between code, is invariant across code transformations. Intuitively, for two operations that share a data or control dependence, there must exist a path that sequentially executes them, which implies that the second operation is reachable from the first one. The compiler must ensure that no dependences are broken if it attempts to modify the relative order or reachability relations of two operations. However, precisely identifying all dependences is very hard because of the imprecise pointer analysis, so the compiler must assume that operations may have dependences unless it can prove otherwise [10]. For example, loop-invariant code motion only moves statements whose definitions are all outside the loop, otherwise, the program may not work correctly [10]. Therefore, code reachability relations are invariant across binaries that are compiled from the same codebase. Our study in Section 3.2 also confirms our insight. Based on the above insight, our key idea is to leverage the stability of code reachability to manipulate the program execution such that deep code logic can also be covered in a consistent way. Since the set of code that can reach a specific program point does not change across compiler transformations, our method constrains the execution within this set. In Section 5, we formally prove that our method guarantees consistent runtime behaviors between semantically similar but synthetically different code fragments.

We implement a prototype in C/C++ and conduct a large-scale evaluation with representative datasets containing 820,021 functions compiled from 16 real-world projects with different compilation settings. Experimental results show that ARCTURUS achieves an average precision of 87.8% with 100% block coverage, outperforming compared methods by 38.4%, on average. In addition, it takes only 0.15 second to process one function, on average, indicating that ARCTURUS is efficient for practical use. The extensive case studies further demonstrate its practical applications in known vulnerability detection and binary version identification.

In summary, we make the following contributions:

- We develop a dynamic binary similarity analysis framework that can achieve both complete code coverage and precise analysis results by using a novel reachability-guided emulation technique.
- We formally prove that the reachability-guided emulation leads to the same execution results among semantically equivalent binaries.
- We implement a prototype ARCTURUS and evaluate it on 820,021 functions from 16 real-world projects. Experimental results show ARCTURUS achieves a precision of 87.8% in function matching, which outperforms the state-of-the-art methods by 38.4%, on average. In addition, it takes only 0.15 second to fully cover one function, on average.

2 PROBLEM STATEMENT

In this article, we focus on measuring the semantic similarity between binary functions to determine if they are compiled from the same code base. In our context, semantics is a particular case that concerns input/output behaviors of the program execution [24, 63, 65], which is defined as follows:

Definition 2.1 (Semantics). The semantics of a binary function is defined as its behaviors in terms of input and output values. An input value is any variable or memory content that influences the output values. An output value is any variable, memory content, and call to library functions that can influence the external environment.

For example, for a specific binary function, its parameter values are its inputs, while values written to the heap memory are its outputs.

We define the **semantic similarity** for two functions in regard to their input-output relations as follows:

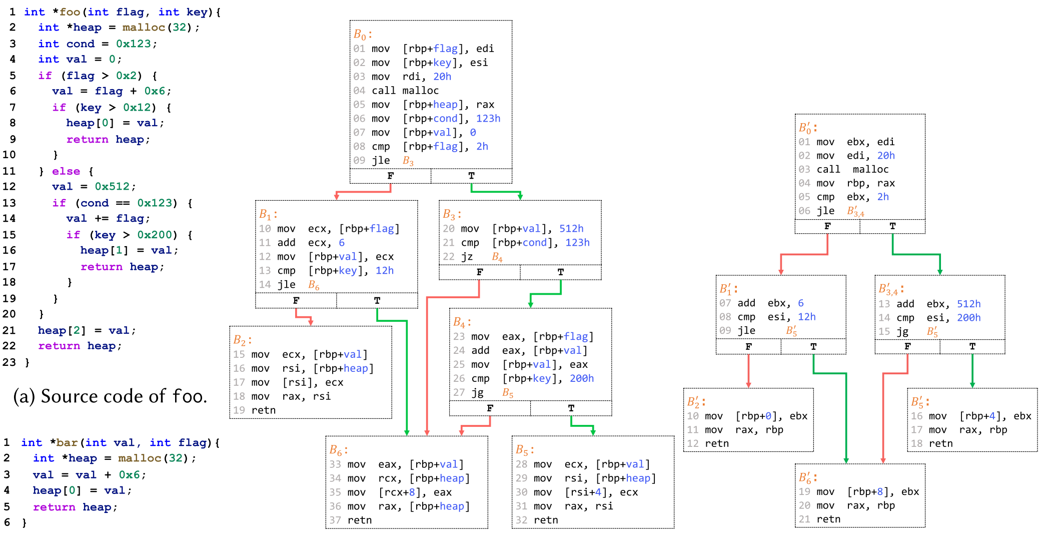
Definition 2.2 (Semantic Similarity). Two binary functions are said to be semantically similar if their semantics (input/output) values have intersections (similar).

We state the problem of **binary similarity analysis** that we want to solve in this article as follows:

Definition 2.3 (Binary Similarity Analysis). Given two binary functions F_1 and F_2, we aim to extract and compare their semantics S_1 and S_2 such that the similarity between S_1 and S_2 is maximized if F_1 and F_2 originate from the same source code, and minimized otherwise.

3 MOTIVATION

In this section, we use an example to illustrate the limitations of existing dynamic methods (Section 3.1), our insight (Section 3.2), and technique (Section 3.3). Figure 1(a) and 1(b) show the source code of functions `foo` and `bar`. They both have the functionality of allocating a chunk of heap memory, writing values to the heap, and returning a pointer to the heap memory. However, their semantics are different in that `foo` writes values based on conditions, while `bar` writes



```
1  int *foo(int flag, int key){
2    int *heap = malloc(32);
3    int cond = 0x123;
4    int val = 0;
5    if (flag > 0x2) {
6      val = flag + 0x6;
7      if (key > 0x12) {
8        heap[0] = val;
9        return heap;
10     }
11   } else {
12     val = 0x512;
13     if (cond == 0x123) {
14       val += flag;
15       if (key > 0x200) {
16         heap[1] = val;
17         return heap;
18       }
19     }
20   }
21   heap[2] = val;
22   return heap;
23 }
```

(a) Source code of `foo`.

```
1  int *bar(int val, int flag){
2    int *heap = malloc(32);
3    val = val + 0x6;
4    heap[0] = val;
5    return heap;
6  }
```

(b) Source code of `bar`.

```
B0:
01 mov  [rbp+flag], edi
02 mov  [rbp+key], esi
03 mov  rdi, 20h
04 call malloc
05 mov  [rbp+heap], rax
06 mov  [rbp+cond], 123h
07 mov  [rbp+val], 0
08 cmp  [rbp+flag], 2h
09 jle  B3

B1:
10 mov  ecx, [rbp+flag]
11 add  ecx, 6
12 mov  [rbp+val], ecx
13 cmp  [rbp+key], 12h
14 jle  B6

B2:
15 mov  ecx, [rbp+val]
16 mov  rsi, [rbp+heap]
17 mov  [rsi], ecx
18 mov  rax, rsi
19 retn

B3:
20 mov  [rbp+val], 512h
21 cmp  [rbp+cond], 123h
22 jz   B4

B4:
23 mov  eax, [rbp+flag]
24 add  eax, [rbp+val]
25 mov  [rbp+val], eax
26 cmp  [rbp+key], 200h
27 jg   B5

B6:
33 mov  eax, [rbp+val]
34 mov  rcx, [rbp+heap]
35 mov  [rcx+8], eax
36 mov  rax, [rbp+heap]
37 retn

B5:
28 mov  ecx, [rbp+val]
29 mov  rsi, [rbp+heap]
30 mov  [rsi+4], ecx
31 mov  rax, rsi
32 retn
```

(c) CFG of `foo` compiled by Clang -O0.

```
B'0:
01 mov  ebx, edi
02 mov  edi, 20h
03 call  malloc
04 mov  rbp, rax
05 cmp  ebx, 2h
06 jle  B'3,4

B'1:
07 add  ebx, 6
08 cmp  esi, 12h
09 jle      B'5

B'3,4:
13 add  ebx, 512h
14 cmp  esi, 200h
15 jg   B'5

B'2:
10 mov  [rbp+0], ebx
11 mov  rax, rbp
12 retn

B'5:
16 mov  [rbp+4], ebx
17 mov  rax, rbp
18 retn

B'6:
19 mov  [rbp+8], ebx
20 mov  rax, rbp
21 retn
```

(d) CFG of `foo` compiled by GCC -O3.

Fig. 1. Motivating example. Both the instruction addresses and the block names are ordered according to the actual code layout in binaries. The stack offsets are represented by the corresponding variable names. The `flag` and `key` parameters are stored in the `edi` and `esi` registers, respectively.

values purely based on the input arguments. Figure 1(c) and 1(d) show the CFGs and part of the assembly code for the `foo` when compiled with different compilation settings. The semantically equivalent basic blocks (i.e., B_m and B'_n) are marked with the same subscript. $B'_{3,4}$ corresponds to the combination of B_3 and B_4, because the tautology condition at Line 13 in Figure 1(a) is removed.

3.1 Limitations of Existing Methods

Native Execution. To measure the similarity of two functions, naive dynamic methods directly execute them with identical inputs to capture their semantics [40, 80]. However, these methods still suffer from the coverage issue [80] and can hardly distinguish programs that have similar shallow paths [70] (e.g., error handling code) but with different deeply embedded behaviors [48]. Therefore, they could draw the wrong conclusion that two binary functions are from the same code base.

Example 3.1. Consider the two functions in Figure 1(a) and 1(b). Suppose their two arguments are assigned with `0x10` and `0x20` for the execution, triggering a similar path, i.e., Lines 2–10 in Figure 1(a) and Lines 2–4 in Figure 1(b). Naive methods might conclude that `foo` and `bar` are similar, which is not true, because the executed paths are unable to capture semantics of all code in the `bar`.

Forced Execution. Some dynamic approaches try to increase the code coverage by overriding the intended program logic [17, 28, 65] according to the CFG structures. For example, the state-of-the-art technique BLEX [28] works by repeatedly executing the first un-executed instruction until every instruction has been executed at least once.

However, since CFGs could be radically modified by code transformations, these approaches cannot guarantee consistent execution results even for semantically similar code. Therefore, their

Table 1. Program States before Executing B_6/B'_6 (State@B_6/B'_6)

BLEX				ARCTURUS			
Clang -O0		GCC -O3		Clang -O0		GCC -O3	
$\langle B_0, B_1, B_2\rangle$	–	$\langle B'_0, B'_1, B'_2\rangle$	–	$\langle B_0, B_1, B_2\rangle$	–	$\langle B'_0, B'_1, B'_2\rangle$	–

$\langle B_3, B_6\rangle$	`val = 0x512`	$\langle B'_{3,4}, B'_6\rangle$	`val = 0x612`
			`key = 0x100`
				$\langle B_0, B_1, B_6\rangle$	`flag = 0x10`
					`val = 0x16`
					`key = 0x20`
						$\langle B'_0, B'_1, B'_6\rangle$	`flag = 0x10`
							`val = 0x16`
							`key = 0x20`

$\langle B_4, B_6\rangle$	`val = 0x200`
	`key = 0x100`						

B_5	–	B'_5	–	$\langle B_0, B_3, B_4, B_5\rangle$	–	$\langle B'_0, B'_{3,4}, B'_5\rangle$	–

$\langle B_i, B_j\rangle$ denotes one executed path that goes through B_i and B_j, respectively.

accuracy is even lower than the native execution-based methods; our evaluation shows that BLEX can only achieve an accuracy of 61% while the native one can reach 91%.

Example 3.2. To compare two binary functions in Figure 1(d) and 1(c) that are compiled from `foo` with different configurations, BLEX first initializes their arguments (suppose to be `0x10` and `0x20`) and executes the two functions. Table 1 shows paths that are executed by BLEX and variable values (behaviors) before executing B_6/B'_6 for two CFGs. For the CFG in Figure 1(c), after triggering the default path $\langle B_0, B_1, B_2\rangle$, BLEX starts the execution from B_3. It assigns the uninitialized variable *cond* with a concrete value (we use `0x100` in this example) to ensure the continuation of the execution. The final path is $\langle B_3, B_6\rangle$, and the value of *val* before executing B_6 is `0x512`. After that, BLEX executes from B_4 and B_5 separately to cover the remaining code. However, for Figure 1(d), although B'_0, B'_1, B'_2, and B'_5 are covered in the same way, the value of *val* before B'_6 is `0x612`, which is different from that in Figure 1(c). As a result, the two functions might not be matched correctly.

Other methods that achieve full code coverage by selecting paths with some self-defined path exploration strategies [17, 35, 64, 94] also share similar limitations with BLEX, as illustrated above. They do not provide any guarantee for the runtime behavior consistency between two semantically similar functions, which might severely affect the similarity analysis results.

3.2 Insight

Given the limitations of existing methods discussed above, the main challenge to a robust similarity analysis technique is how to achieve high code coverage and behavior consistency between semantically similar code simultaneously.

Our technique is inspired by two key insights. First, *the reachability relations reflect control or data dependence between code.* For two operations that share a dependence, there must exist a path that sequentially executes them so the second operation is reachable from the first one.

Example 3.3. In Figure 1(c), instructions in B_6 have data dependence with instructions in B_3. Thus, B_6 is reachable from B_3.

Second, *code transformations must not change code dependences, thus the code reachability is stable across semantically similar binaries.* The compiler must ensure that no dependencies are broken if it attempts to modify the reachability relation between two operations. However, precisely identifying all dependencies is very hard because of the imprecise pointer analysis, so the compiler must assume that operations may depend on each other unless it can prove otherwise [10].

Example 3.4. After the optimization (Figure 1(d)), although B_3 is merged into $B'_{3,4}$, B'_6 is still reachable from $B'_{3,4}$.

We define the code reachability on CFGs as follows:

Definition 3.1 (Code Reachability). A CFG node v_i is said to be reachable from another CFG node v_j if there exists a path that starts from v_j and goes through v_i, denoted as $v_j \rightsquigarrow v_i$. Otherwise, v_i is unreachable from v_j, denoted as $v_j \not\rightsquigarrow v_i$.

Additionally, if $v_j \rightsquigarrow v_i$, then a given program point p inside v_i is also reachable from v_j, denoted as $v_j \rightsquigarrow p$.

Example 3.5. In Figure 1(c), there exist two paths from B_3 to B_6 and one path from B_4 to B_6, which means that B_6 is reachable from B_3 and B_4. In Figure 1(d), B_6' corresponds to B_6 in Figure 1(c). After compiler optimizations (Figure 1(d)), although B_3 and B_4 are merged into $B_{3,4}'$, B_6' is still reachable from $B_{3,4}'$, which indicates that their reachability relations are still preserved.

To clearly understand our insight, we conduct a study to measure the stability of code reachability across binaries compiled from different compiler settings. Specifically, we take -O0 compiled (unoptimized) binaries as references and compare the reachability results of -O3 compiled binaries with them. For each CFG node in binaries, we calculate the set of nodes that can reach it and the set of nodes that cannot reach it. To perform the comparison, we leverage the debug information to map basic blocks in binaries with source code lines. If we can find two corresponding nodes in two binaries (line numbers that are deleted in optimized binaries are excluded), then we calculate the ratio of the intersection between the sets of reachable line numbers and the sets of unreachable line numbers.

In the study, we compile all the projects used in our evaluation (details in Table 2) with two compiler versions (GCC v7.5.0 and Clang v8.0.0) and two optimization levels (-O0 and -O3). We choose these two optimization levels because they are thought to be able to maximize differences between binaries [66]. We also disable function inlining to measure only intra-procedural reachability. Figure 2 shows the comparison results between -O0 and -O3 optimized binaries. On average, the reachability set intersection ratio between different binaries is as high as 97.68%, which matches up with our insight. After manually analyzing false cases, we found that code motion is the prominent reason leading to mismatches. Code motion affects reachability by moving statements upwards or downwards. We only discuss basic block-level code motion here, since moving code inside the block does not affect block-level reachability. There are mainly two kinds of statements that are usually moved. First, the compiler can move variable assignments. It usually happens when a variable is assigned in the function entry but used in later code. Under this situation, the compiler will move the variable assignment just before its uses, causing a mismatch of reachable sets. Second, if a function has multiple return statements, then the compiler might merge them and transform the original statements into assignments, thus causing reachable sets mismatching.

3.3 Our Technique

Based on the above insight, we propose a new technique that leverages the stability of code reachability to manipulate the program execution such that deep code logic can be covered in a consistent way. Since the set of code that can reach a specific program point does not change across compiler transformations, our key idea is to constrain the execution within this set. In Section 5, we formally prove that our method guarantees consistent runtime behaviors between semantically similar but synthetically different functions.

Example 3.6. We use the motivating example to demonstrate our technique. As shown in Table 1, for the CFG in Figure 1(c), after covering the default path $\langle B_0, B_1, B_2 \rangle$, ARCTURUS sets one uncovered basic block as the target according to the reverse topological order of the CFG. By setting B_6 as the target, it starts executing from B_0 and performs reachability-guided emulation to cover

Table 2. Projects Adopted in the Evaluation

Programs	LOC	# Function	# Block
ImageMagick-7.0.10	570,420	3,264	190,999
binutils-2.35	4,547,167	2,750	73,525
busybox-1.33.0	198,259	2,710	88,406
coreutils-8.30	749,329	1,313	42,292
diffutils-3.7	178,327	3,322	8,942
gpac-1.0.1	785,503	8,753	394,274
htslib-1.10.2	62,173	1,278	31,672
libarchive-3.4.4	232,147	1,823	34,189
libxml2-2.9.9	538,015	2,095	88,434
nginx-1.18.0	139,459	2,299	31,111
openssl-1.0.1u	376,774	4,523	142,888
putty-0.74	143,314	2,103	34,651
sqlite-3.35.0	213,820	2,442	93,105
tcpdump-4.10.0	161,929	1,691	47,863
transmission-3.00	368,670	2,812	38,424
vim-8.2	866,141	5,571	238,596

The second column indicates the size of the source code. The third and fourth columns show the average number of functions and basic blocks in the compiled binaries.

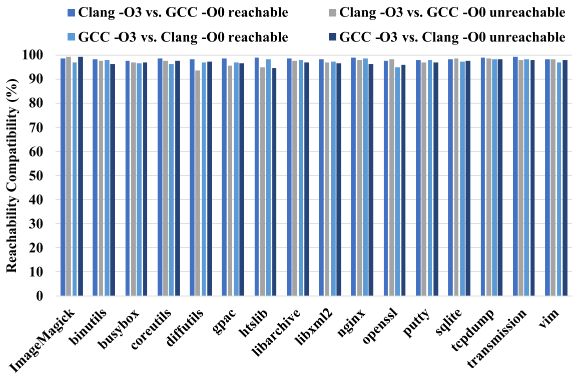


Fig. 2. Reachability compatibility between -O0 and -O3 optimized binaries.

B_6. In instruction 14, the default path goes through the false branch to B_2, which is unreachable to the target block B_6. Therefore, ARCTURUS directly sets instruction 33 to be the next one for executing and leaves the execution context untouched. By doing this, the executed path will be $\langle B_0, B_1, B_6 \rangle$, and the variable values before B_6 is (`flag`=0x10, `val`=0x16, `key`=0x20). The final output of the above path will contain a memory write to `heap[1]` with the value of `val` variable (0x16). Similarly, to cover B'_6 in Figure 1(d), the executed path will be $\langle B'_0, B'_1, B'_6 \rangle$ with the output that also contains a memory write to `heap[1]` with the value of 0x16. Since the output values of the two functions are the same, ARCTURUS can match them correctly.

4 DESIGN

4.1 Overview

To obtain the similarity score between two binary functions, ARCTURUS mainly works in three steps: pre-processing, emulation, and comparison. Figure 3 shows the workflow of ARCTURUS. Both the target and reference binary code are processed in the same way. To compare the target

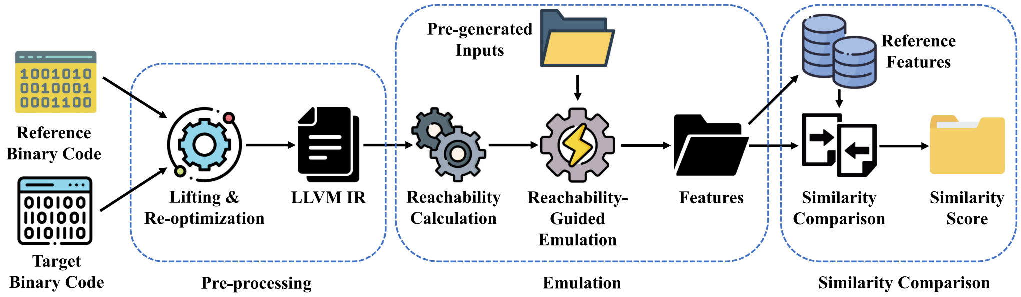


Fig. 3. Workflow of ARCTURUS.

binary function with the reference one, assuming the reference code has been analyzed, the target code is analyzed as follows:

- **Pre-processing & Lifting**. The target binary code is lifted into the LLVM [5] **intermediate representation (IR)** by using a binary lifter [18]. ARCTURUS further adopts sophisticated optimizations provided by LLVM to re-optimize the IR. The purpose is to reduce the number of instructions significantly to boost the emulation process.
- **Emulation**. ARCTURUS captures various kinds of runtime behaviors during the emulation and uses them as semantic features for similarity comparison. It first performs the reachability calculation for all intra-procedural blocks of each function. A backward graph reachability algorithm is used to compute the set of blocks that can reach each basic block. Then, the emulation engine is invoked to execute the target code and collect its runtime behaviors, which are stored as the semantic features of the function. The engine emulates each function using a set of predefined parameter values. It covers all intra-procedural blocks by dynamically forcing a set of branch outcomes under the guidance of reachability.
- **Similarity Comparison**. The extracted features of the target code are compared with reference features extracted from reference functions in the function pool to search for the most similar one. The similarity score between two functions is calculated by comparing their semantic feature sets using Jaccard containment similarity [28, 40, 80]:

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}, \tag{1}$$

  where S_1 and S_2 represent the semantic features of the two functions, respectively. The calculated score is a numeric value ranging from 0 to 1. Two functions are considered more similar when the score is closer to 1. ARCTURUS generates a list of similar functions by ranking the Jaccard Index in descending order.

The remainder of this section is organized as follows: We first introduce semantic features used to compare binary functions. Then, we discuss the emulation engine. At last, we discuss the algorithm of reachability-guided emulation.

4.2 Semantic Features

ARCTURUS captures various kinds of runtime behaviors (input/output values) during the emulation and uses them as semantic features for similarity comparison. However, not all runtime values are strongly related to program functionalities and can be easily altered by code transformations. For example, the code in Figure 1(c) heavily uses the stack memory to load and store intermediate values for other operations. By contrast, in Figure 1(d), the compiler optimized the code such that the assembly code uses registers to access all the variables instead of the stack frame. Despite this, there exist behaviors that cannot be changed by the compiler and are suitable for revealing the real program semantics [28, 40, 80]. For example, in both Figure 1(c) and 1(d), the calls to the heap allocation function and the values written to the heap memory are the same. Specifically, the following behaviors are recorded as the semantic features of the target function to alleviate the syntactic and structural gaps caused by code transformations during emulation:

Heap/global memory access. The read/write values of heap/global memory reflect the semantics of the code and are unlikely to be influenced by code transformations. For a binary function, the global variables are stored in data sections (e.g., `.data`), while the heap memory is dynamically allocated by library functions (e.g., `malloc`). Additionally, ARCTURUS regards memory regions pointed to by pointer parameters as the heap memory.

Example 4.1. Both Figure 1(c) and 1(d) c-ontain instructions that write values to the allocated heap memory. In Figure 1(c), the address of the heap memory is stored in `rcx` and instruction 35 in B_6 writes the value of `eax` to `[rcx+8]`. Upon emulating instruction 35, ARCTURUS will record the value of `eax` and the address of `[rcx+8]` as the features.

Invoked library functions. Symbol names of library functions persist in stripped binaries, because they are necessary for resolving external function calls. Therefore, ARCTURUS records library functions' names as the code feature at their call sites.

Example 4.2. After binary lifting, Instruction 4 in Figure 1(c) will be translated to a statement like `%ret=call malloc(0x20)`. ARCTURUS records a single name `malloc` as the semantic features.

Function return value. The function return values are also unlikely to be influenced by code transformations. For internal functions, the return value is stored into specific registers in function bodies according to the calling convention of the target architecture, e.g., integer return values on x64 are returned in `RAX` if 64 bits or less. ARCTURUS models behaviors of common external functions and takes possible alias library functions as the same under some conditions (e.g., `puts` and `printf`). Therefore, ARCTURUS is able to retrieve function return values at all call sites.

Example 4.3. For the Instruction 4 in Figure 1(c), its call site looks like `%ret=malloc(0x20)`. ARCTURUS assigns a heap address to `%ret` and records the value of `%ret` as the semantic features.

4.3 Emulation Engine

According to the formal semantics of LLVM statements [98], ARCTURUS develops an emulation engine to reason about program execution without interfering with the underlying operating system. Compared with concrete execution that relies on real CPUs, the advantages of emulation are twofold. First, emulation is much more efficient than heavy-weight dynamic binary instrumentation [55], because emulation omits many system details and discards runtime binary translation. In our experiments, ARCTURUS only takes 0.03 s to emulate one function, but concrete execution needs 11.37 s to process one function. Second, emulation allows us to take full control of the execution procedure and obtain more comprehensive runtime information. Different from a symbolic interpretation [52], our emulation engine assigns all variables with concrete values to capture program execution semantics. A handler is defined for each LLVM statement in terms of how they update intermediate values and memory. ARCTURUS invokes different handlers for statements during emulation and updates the program counter according to the handler results.

Memory Management. ARCTURUS defines its own memory model to handle memory access. It partitions the whole memory into three disjoint areas, i.e., global memory, stack memory, and heap memory. Before emulation, global memory is initialized according to the global variables declared in LLVM IR. It maintains a stack to record the local execution context of each call site. Once a function is called, a clean execution environment is pushed to the stack. All local variables are allocated on the stack memory region. After the function returns, it cleans the stack memory allocated by the local execution context. Heap memory is dynamically allocated based on a call-to-allocation function (e.g., `malloc`). Since the emulation process usually crafts random values as the function input, it is possible that some memory accesses do not have pre-allocated values and become invalid [64]. To solve this problem and make the emulation continue, ARCTURUS adopts an "on-demand memory allocation" strategy; that is, once an invalid memory address is accessed, it allocates new memory on the accessed address and fills it with a pre-defined dummy value.

Recursion and Loops. The emulation process could also be trapped into recursive calls and infinite loops. To prevent this, ARCTURUS monitors the executed traces of the target program to detect recursion and loops. To prevent recursive function calls, ARCTURUS checks the call stack at each call site. Upon calling a function, it checks if the callee function exists in the call stack. If so, then it directly skips this call site. For loops, ARCTURUS only unrolls them once to prevent getting stuck into infinite loops. Before emulation, ARCTURUS analyzes all loops to extract their loop headers and exit nodes [69]. During emulation, ARCTURUS maintains a loop context to track all loop accesses; once a loop is iterated for more than one time, ARCTURUS will break the loop by taking the exit node.

Internal Function. Since the function body of the internal function is present in the lifted code, ARCTURUS is able to jump to the function entry at its call sites and emulate statements one-by-one using the technique described above until encountering the return instruction.

External Function. For external functions, since their definitions are unknown, we cannot emulate them directly. Therefore, ARCTURUS wraps some common external functions to extract semantic features better. For common alias library functions such as "`printf`" and "`__printf_chk`," ARCTURUS models them in the same way to avoid different emulation results. To support custom memory management, ARCTURUS wraps memory allocation and de-allocation functions (such as `malloc` and `free`) to manage heap memory. For common I/O functions such as `getc`, we return a sequence of pseudo-random numbers to capture their calling "contexts," meaning that we are able to distinguish between different call sites of the same function. For other functions that are not modeled, we return a default dummy value.

Exception Handling. ARCTURUS performs emulation with possible invalid inputs and paths, so it is possible to trigger exceptions such as null pointer deference. Since ARCTURUS leverages an emulation engine to traverse the code instead of the actual execution on CPUs, all exceptions can be suppressed so the execution continues. Additionally, exceptions can also produce valuable semantic features. For example, for the null pointer dereference, ARCTURUS will record that the code reads memory from address 0x0.

4.4 Reachability-guided Emulation

The reachability-guided emulation process is described in Algorithm 1. It takes the LLVM IR of the target function $\mathcal{F}$ as the input and outputs its semantic set $\mathcal{S}_{\mathcal{F}}$.

Initialization. ARCTURUS initializes the function inputs by assigning random values to $\mathcal{F}$'s parameters. In the lifted code, $\mathcal{F}$'s parameters are declared as an ordered list of values. ARCTURUS uses a pseudo-random number generator to generate a deterministic sequence of values, which guarantees that each position in the sequence always has the same value each time. Therefore, ARCTURUS can ensure that parameters in the same positions of the ordered list are always assigned with the same initial value (Line 1). Since precise type information is lost in the binary code, ARCTURUS simply assigns integer values to all parameters, and possible invalid memory access is handled in the way described in Section 4.3.

ALGORITHM 1: Reachability-guided Emulation

```
   Input: Target Function $\mathcal{F}$
   Output :Function semantic feature set $\mathcal{S_F}$
1  $\mathcal{I}$ ← Init_Input()                              ▷ generate input values for $\mathcal{F}$
2  $\mathcal{M_R}$ ← Calculate_Reachability($\mathcal{F}$)    ▷ Map of basic blocks to their reachable blocks
3  $\mathcal{Q}$ ← Reverse_Topological_Sort($\mathcal{F}$)    ▷ sort basic blocks of $\mathcal{F}$ in reverse topological order
4  while $\mathcal{Q} \neq \varnothing$ do
5      $\mathcal{B_T}$ ← Pop_Front($\mathcal{Q}$)
6      while $\mathcal{B_T}$ has been covered before do
7          $\mathcal{B_T}$ ← Pop_Front($\mathcal{Q}$)          ▷ get the first uncovered block in $\mathcal{Q}$
8      $\mathcal{E}$ ← Init_Env()                             ▷ initialize execution environment
9      $\mathcal{S}$ ← the first statement of $\mathcal{F}$
10     while $\mathcal{S}$ is legal do
11         $\mathcal{S_F}, \mathcal{E}$ ← $\mathcal{S_F}$ ∪ emulate($\mathcal{S}, \mathcal{I}, \mathcal{E}$)   ▷ emulate and collect features from $\mathcal{S}$
12         if $\mathcal{S}$ is branch statement then
13             $\mathcal{B_O}$ ← Get_Target($\mathcal{S}$)     ▷ obtain original jump target block
14             if $\mathcal{S} \in \mathcal{F}$ and $\mathcal{B_T}$ has not been covered and $\mathcal{B_O}$ cannot reach $\mathcal{B_T}$ then
15                 $\mathcal{B_N}$ ← Branch_Alter($\mathcal{M_R}$)   ▷ find another target block that can reach $\mathcal{B_T}$
16                 if $\mathcal{B_N}$ is a valid block then
17                     $\mathcal{B_O} \leftarrow \mathcal{B_N}$            ▷ decide to change the branch
18             $\mathcal{S}$ ← first statement of $\mathcal{B_O}$      ▷ branch to new block
19             continue
20         else if $\mathcal{S}$ is termination then
21             break
22         else
23             $\mathcal{S}$ ← next statement
```

Example 4.4. For the assembly code in Figure 1(c), the binary lifter of ARCTURUS will identify `edi` and `esi` as the two parameters. During initialization, ARCTURUS generates two random values and assigns them to the two parameters, respectively.

ARCTURUS calculates the code reachability relations between basic blocks for the function $\mathcal{F}$ with a backward graph reachability algorithm [46] (Line 2). To emulate $\mathcal{F}$, ARCTURUS sorts all intra-procedural basic blocks using the reverse topological order and puts them into a queue $\mathcal{Q}$ (Line 3). According to the reverse topological sort of the CFG, each time, it sets an uncovered block $\mathcal{B_T}$ as the target (Line 7), which should be covered in the following emulation. By covering all basic blocks in a bottom-up fashion, it achieves complete code coverage more quickly.

Example 4.5. In Figure 1(d), the set of reachable blocks for B'_6 is $\left\{B'_6, B'_1, B'_{3,4}, B'_0\right\}$. After the CFG sorting, $\mathcal{Q} = \left\{B'_6, B'_5, B'_2, B'_1, B'_{3,4}, B'_0\right\}$.

Before the emulation starts, it initializes the execution environment (Line 8) by filling the global and local memory regions with declared variables.

Emulation. Then, the emulation starts from the entry point of the function $\mathcal{F}$, which is its first statement. The emulation engine emulates the current statement $\mathcal{S}$ with the input $\mathcal{I}$ and environment $\mathcal{E}$. During emulation, runtime behaviors are collected into $\mathcal{F}$'s semantic feature set

$\mathcal{S}_\mathcal{F}$, and the $\mathcal{E}$ is also updated to handle the following statements (Line 11): To collect semantic features described in Section 4.2, ARCTURUS monitors all program operations, such as memory read/write and function calls. If they satisfy the semantic feature description, then we put related values into the set $\mathcal{S}_\mathcal{F}$.

Example 4.6. For the binary code in Figure 1(c), ARCTURUS emulates it from the first instruction. For instruction 4, ARCTURUS emulates it by allocating a heap memory region with size 0x20 and assigning the address to `rax`. Then, ARCTURUS moves the program counter to the next instruction 5, which stores the value of `rax` to the stack memory `[rbp+heap]`. Moreover, ARCTURUS records the name of `malloc` function and the allocated address as the semantic features.

Line 12–Line 19 illustrates the core steps that allow ARCTURUS to achieve full code coverage and high accuracy at the same time. For the target block $\mathcal{B}_\mathcal{T}$, the algorithm aims to alter the execution path to cover $\mathcal{B}_\mathcal{T}$ by monitoring branch statements. More specifically, if statement $\mathcal{S}$ is a branch statement inside target function $\mathcal{F}$ and $\mathcal{B}_\mathcal{T}$ has not been covered yet, then ARCTURUS checks whether $\mathcal{B}_\mathcal{T}$ is reachable from the default branch outcome $\mathcal{B}_\mathcal{O}$ by querying the reachability map $\mathcal{M}_\mathcal{R}$ (Line 14). If $\mathcal{B}_\mathcal{O}$ can reach $\mathcal{B}_\mathcal{T}$, then no modification is performed, and the execution continues because it has the potential to cover $\mathcal{B}_\mathcal{T}$ next. Otherwise, ARCTURUS tries to find another branch that can reach $\mathcal{B}_\mathcal{T}$ (Line 15).

Example 4.7. For the binary code in Figure 1(d), suppose the two input parameters (`edi` and `esi`) take 0x5 and 0x10, respectively. ARCTURUS first sets the target block $\mathcal{B}_\mathcal{T}$ as B'_6 and executes the function from instruction 1. For the branch statement at instruction 9, since `esi` $< 0x12$, the default branch outcome goes to B'_2, which is unreachable to B'_6. ARCTURUS modifies the branch outcome to be B'_6 because it is reachable to B'_6.

Since the target $\mathcal{B}_\mathcal{T}$ must be reachable from the function entry point, each executed jump instruction must have at least one successor that can reach $\mathcal{B}_\mathcal{T}$ following the algorithm. If s is a conditional jump, then $\mathcal{B}_\mathcal{T}$ must be reachable from one of its successors, so ARCTURUS simply assigns the other successor of $\mathcal{S}$ as the following block to be emulated. The switch statement proposes additional challenges to ARCTURUS since it usually has more than two successor branches. As a result, when the natively executed branches in the switch statement cannot reach the target block, we have to choose from multiple branches to decide which one to take. ARCTURUS currently chooses the last entry in its jump table that can reach the target block as the alternative branch. We argue that such a design is reliable even considering complicated transformations (e.g., nested if-else at -O0 and jump table at -O3). To facilitate the indexing, entries of a switch jump table are arranged in a sequential order in binaries. Therefore, for binaries without special obfuscations, the entry order in a jump table is sequential, even with heavy compiler optimization. Consider a switch statement in source code; there are three possible situations to consider when it is compiled into binaries under different configurations.

- The first situation is that the switch-case or the nested if-else are all compiled into nested branches. Then, no branch has more than two successors. Therefore, ARCTURUS works correctly following the proof in Section 5.
- The second situation is that the switch cases or the nested if-else are all compiled to jump tables. The jump tables in binaries must have been ordered according to the values of cases or conditions. Therefore, ARCTURUS's current strategy also works correctly.
- The third situation is that one of the switch cases or the nested if-else is optimized to the jump table while the other one is optimized to nested branches. In most cases, ARCTURUS's re-optimization step can unify different formats and thus fall back to the first two situations. In rare cases where re-optimization cannot unify the formats, we found that the nested if-else

is still ordered as jump tables. By examining the if-else branches one-by-one, our algorithm will choose the last branch that can reach the target point, which equals finding the last entry in its jump table that can reach the target.

After finding a valid target $\mathcal{B}_{\mathcal{N}}$, it overrides the control-flow by force, changing the jump target from $\mathcal{B}_{\mathcal{O}}$ to $\mathcal{B}_{\mathcal{N}}$ (Line 17). The emulation terminates when all intra-procedural blocks are covered, and the feature set $\mathcal{S}_{\mathcal{F}}$ will be used to calculate the similarity score between two functions.

5 FORMAL ANALYSIS OF CORRECTNESS

In this section, we prove that the reachability-guided emulation guarantees the same execution results for two semantically equivalent functions, which is an ideal scenario of semantically similar functions (functions compiled from the same source code). In real cases, we usually have to handle functions that are not totally semantically equivalent. So, the extracted features are not exactly the same but have some similarities. At a high-level, our proof shows that every time a branch outcome is changed, the execution paths for two semantically equivalent program points still have consistent runtime behaviors among semantically equivalent binaries.

5.1 Preliminaries

We first provide basic definitions and assumptions that we use throughout the proof. To present the proof clearly, we introduce the following notations and terminologies:

- F_1 and F_2 are two binary functions that are compiled from the same source code (semantically equivalent), denoted as $F_1 \equiv F_2$.
- p_1 and p_2 are two program points inside F_1 and F_2 that correspond to the same program point in the original source code, which are semantically equivalent (aligned), denoted as $p_1 \equiv p_2$.
- $e_i = (v_i, v_j)$ denotes a directed edge in the CFG that goes from v_i to v_j.
- $\mathcal{P} = \langle v_0, v_1, \ldots, v_n \rangle$ is a program path in the CFG, where v_k is a CFG node.
- $e_i = (v_i, v_j) \in \mathcal{P}$ means a program path goes through a CFG edge e_i.
- $p \in \mathcal{P}$ means a path $\mathcal{P}$ goes through a program point p.
- $\mathcal{P}_{F_1}$ and $\mathcal{P}_{F_2}$ are two paths that are obtained from natively executing F_1 and F_2 with the same program inputs.

Definition 5.1 (Reachability Set). For a given CFG node N, its reachable set $\mathcal{R}_N$ and unreachable set $\mathcal{U}_N$ are defined as follows:

$$\mathcal{R}_N = \{v \mid v \rightsquigarrow N\}$$
$$\mathcal{U}_N = \{v \mid v \not\rightsquigarrow N\},$$

where all nodes in $\mathcal{R}_N$ and $\mathcal{U}_N$ are intra-procedural nodes, and the two sets are disjointed.

Additionally, a given **program point** p inside N has the same reachable set and unreachable set as N, denoted as $\mathcal{R}_p$ and $\mathcal{U}_p$.

Example 5.1. In Figure 4, for the CFG node T, the corresponding $\mathcal{R}_T$ and $\mathcal{U}_T$ are $\{A, B, D, T\}$ and $\{C, E, F, G, H, I, J, K\}$, respectively.

Definition 5.2 (Border Edge). For a given program point p, we define **Border Edge** in regards to p as $E_b^p = \{(v_i, v_j) \mid (v_i \in \mathcal{R}_p \wedge v_i \neq p \wedge v_j \in \mathcal{U}_p)\}$

Border edges connect nodes in different reachability sets of the block p belongs to.

Example 5.2. In Figure 4, $E_1 = (A, C) \in E_b^T$, while $E_2 = (A, B) \notin E_b^T$.

We now formally prove that border edges can only start from $\mathcal{R}_p$ and end at $\mathcal{U}_p$, but not vice versa.

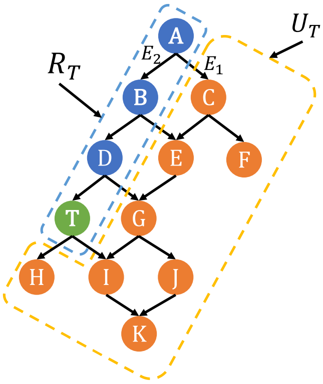


Fig. 4. $\mathcal{R}_T$ and $\mathcal{U}_T$ of the target block T.

Lemma 5.1. *For a given program point p.*

$$\forall (v_i, v_j) \in E_b^p \Rightarrow v_i \in \mathcal{R}_p \land v_j \in \mathcal{U}_p$$

Proof. We prove it by contradiction. Suppose there exists an edge (v_i, v_j) such that $v_i \in \mathcal{U}_p$ and $v_j \in \mathcal{R}_p$. Since $v_j \in \mathcal{R}_p$, $v_j \rightsquigarrow p$. Suppose the path starts from v_j that reaches p is $\mathcal{P}_1 = \langle v_j, \ldots, p \rangle$, then the path $\mathcal{P}_2 = \langle v_i, v_j, \ldots, p \rangle$ starts from v_i and reaches p. Therefore, $v_i \rightsquigarrow p$, and $v_i \notin \mathcal{U}_p$, a contradiction. □

This lemma implies there does not exist a path that starts from a block inside $\mathcal{U}_p$ and goes through a block inside $\mathcal{R}_p$. Therefore, if any path goes through any border edge, then it must not go through p.

Example 5.3. In Figure 4, if a path goes through E_1, then it would never go through T.

Now, we state the key assumption that guides and motivates our approach. We observe that this assumption holds for most binaries, in general (see the study in Section 3.2).

Assumption 5.1. *The CFG nodes inside $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$ have the same semantics given that $p_1 \equiv p_2$.*

This assumption means that two execution paths inside $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$ should have the same runtime behaviors given that they are executed from the aligned program points (e.g., function entry) with the same initial state (e.g., same initial variable definitions and memory states). We can also conclude that $\mathcal{U}_{p_1}$ and $\mathcal{U}_{p_2}$ have the same semantics, since $F_1 \equiv F_2$.

5.2 Proof

We establish the following lemmas to prove the correctness of our approach:

Lemma 5.2. *For a given program point p and a program path $\mathcal{P}$ that terminates without aborting.*

$$\forall e_i \in \mathcal{P}, e_i \notin E_b^p \Rightarrow p \in \mathcal{P}$$

Proof. Let the entry node be v_0, it is evident that $v_0 \in \mathcal{R}_p$, since the entry node can reach all intra-procedural nodes.

We prove it by contradiction. Suppose a path $\mathcal{P} = \langle v_0, v_1, \ldots, v_k \rangle$ that does not go through any border edges and does not contain p. Since $\mathcal{P}$ does not go through any border edges, all nodes of $\mathcal{P}$ must belong to $\mathcal{R}_p$, thus $v_k \in \mathcal{R}_p$.

Since v_k is an exit node, it does not have any children and there does not exist a path that starts from v_k and goes through p. Therefore, $v_k \not\rightsquigarrow p$ and $v_k \notin \mathcal{R}_p$, a contradiction. □

In other words, if a path does not go through any border edges, then it must go through p.

Example 5.4. In Figure 4, the only path that does not go through border edges is $\langle A, B, D, T\rangle$, which goes through T.

Lemma 5.3. $\exists e_1 \in \mathcal{P}_{F_1}, e_1 \in E_b^{p_1} \Rightarrow \exists e_2 \in \mathcal{P}_{F_2}, e_2 \in E_b^{p_2}$.

Proof. We prove it by contradiction. $\mathcal{P}_{F_1}$ and $\mathcal{P}_{F_2}$ have the same semantics, as they are executed from the function entry nodes under the same inputs. Suppose $\mathcal{P}_{F_1}$ does not go through any border edge of p_1, while $\mathcal{P}_{F_2}$ goes through a border edge of p_2. According to Lemmas 5.1 and 5.2, $\mathcal{P}_{F_1}$ goes through p_1, while $\mathcal{P}_{F_2}$ does not go through p_2. As a result, $\mathcal{P}_{F_1}$ contains the semantics in p_1, while $\mathcal{P}_{F_2}$ does not contain the semantics in p_2. They have different semantics, a contradiction. □

In other words, given the same inputs, if one of the natively executed paths in F_1 and F_2 ($F_1 \equiv F_2$) goes through a border edge, then the other one must also go through a border edge. Intuitively, the path that goes through the border edge contains the semantics in the unreachable set. In contrast, the path's runtime behaviors that do not go through the border edge only reflect the semantics inside the reachable set. Therefore, as long as two paths have the same semantics, all of them or none of them should go through border edges with respect to p_1 and p_2.

Lemma 5.4. *If $\mathcal{P}_{F_1}$ and $\mathcal{P}_{F_2}$ go through border edges, then the two program points before the border edges can be aligned and have the same program state.*

Proof. According to Assumption 5.1, for two semantically equivalent program points p_1 and p_2, execution paths of $\mathcal{P}_{F_1}$ and $\mathcal{P}_{F_2}$ inside $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$ have the same final state. Therefore, the two program points before the two paths going through border edges have the same program state and can be aligned. □

Based on the above lemmas, we can now prove two theorems of our approach.

Theorem 5.1. *Algorithm 1 produces the same execution results for two paths that cover p_1 and p_2 ($p_1 \equiv p_2$).*

Proof. There are two cases to consider.

- If both p_1 and p_2 can be covered with native execution, then the two paths must have the same behaviors.
- If natively executing F_1 goes through a border edge (v_i, v_j). According to Lemma 5.3, the natively executed path of F_2 must go through a border edge (denoted as (v_p, v_q)). According to Lemma 5.4, two program points before going through (v_i, v_j) and (v_p, v_q) can be aligned and have the same program state. Our technique changes the branch outcomes in v_i and v_p to blocks inside the reachable sets.
  - If Both v_i and v_p have only one successor inside $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$, then modifying the branch outcomes in v_i and v_p can be viewed as natively executing from the same program point (ends of v_i and v_p) with the same program state. Therefore, the later execution paths are also equivalent.
  - Optimized switch statements could make one or both of v_i and v_p have more than one successor inside $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$. Since entries of a switch jump table are arranged in a sequential order in binaries, choosing the last entry still ensures consistent execution paths (more discussions about switch are presented in Section 4.4).

We have shown that, whenever the execution paths go through the border edges, our technique can still ensure aligned program points having the same program states. Therefore, the execution results of the two paths must be the same. □

Theorem 5.2. *Algorithm 1 produces the same execution results for two semantically equivalent functions F_1 and F_2.*

Proof. According to Theorem 5.1, Algorithm 1 guarantees the same execution results for two aligned program points. Since it covers all program points in F_1 and F_2, all aligned program points in F_1 and F_2 are covered with consistent execution paths and produce the same runtime behaviors. Therefore, our technique guarantees the same execution results for two semantically equivalent functions. □

6 IMPLEMENTATION AND EXPERIMENT SETUP

6.1 Implementation

We have implemented ARCTURUS on our binary translation framework, which consists of over 160k lines of C/C++ code. We use CAPSTONE [16] as the underlying disassembler to translate raw binary bytes into assembly and lift the assembly code into LLVM IR. Our similarity comparison, including the emulator, is written with over 8,000 lines of C/C++ code. The emulation engine is implemented on top of the LLVM instruction interpreter. Currently, ARCTURUS supports binaries on the x64 platform. It can be easily extended to support other architectures due to the cross-platform feature of LLVM IR.

6.2 Experiment Setup

We evaluate ARCTURUS on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64 GB of memory running Ubuntu 16.04 LTS.

Dataset Collection. The dataset used in the experiment consists of **16** programs (details in Table 2) that are: (i) widely used in evaluating previous work (e.g., `Coreutils` [79, 92]), (ii) adopted by publicly available datasets like `BINKIT` [49] (e.g., `OpenSSL`), and (iii) common in real-world scenarios (e.g., `vim`). Besides, the **16** programs adopted in the evaluation also have sufficient diversity in terms of the source code size (ranging from 63 KLOC to 4,547 KLOC) and the functionality (e.g., image processing and authentication) to show both the effectiveness and scalability of ARCTURUS.

For benign code analysis, the projects are compiled with different optimization levels (`O0-O3`), using two modern compilers of different versions, i.e., GCC v7.5.0/4.8.1 and Clang v8.0.0/3.6.2. Moreover, we adopt BinTuner [66] to maximize the binary code syntactic differences (denoted as -`O`*B*), showing the effectiveness of ARCTURUS. BinTuner leverages a guided stochastic algorithm to explore how combinations of compiler optimization passes can obfuscate software. Since its prototype [4] fails to process all the projects, we only use it to search for optimization option sequences for `Coreutils` and `OpenSSL` compiled with GCC, which is in accordance with their paper.

Additionally, for obfuscated code analysis, we use the three strategies provided by OLLVM [7] to handle the projects with the optimization option -O3: (i) **Bogus Control Flow (BCF)**, breaking original basic blocks up by inserting opaque predicates; (ii) **Control Flow Flattening (FLA)**, splitting a function control flow graph into parts and putting them inside an infinite loop with a `switch` structure to maintain the original control flow; and (iii) **Instruction Substitution (SUB)**, replacing the original instruction with functionality equivalent but more complicated ones. Since OLLVM is unable to handle all projects, only seven of them are successfully obfuscated for the experiments.

In total, we compiled 286 binaries (`Coreutils` is compiled into a single binary file) with 820,021 functions for evaluation.

Ground Truth and Metrics. The debug and symbol information of all the aforementioned binaries is stripped for evaluation. To verify the correctness of the experiment results, we compile their extra unstripped copies and leverage the symbol names of functions as the ground truth. Two functions are considered to be correctly matched if they have the same ***symbol name*** [49, 92].

Similar to previous research [17, 26, 28, 29, 80], we adopt *Precision@K* to measure the correctness of function matching between two binaries, i.e., the percentage of target functions whose correct matches are found within the Top K entries in the resulting function list:

$$Precision@K = \frac{|\{\, b \in B \mid n_b \in N_L^K \,\}|}{|B|}, \tag{2}$$

where B is the target function set, n_b is the symbol name of the function b, and N_L^K is the symbol-name set of entries whose similarity scores are ranked within Top K of the resulting function list L (L has been sorted by the similarity scores in descending order).

For example, consider the target binary bin_1 and the reference binary bin_2 that are compiled from the same source code but with different compilation configurations. For a function f in bin_1, we iteratively compare it with all functions in bin_2 and produce the resulting function list L by sorting all the scores. If the symbol name of f is the same as the first function in L, then we will regard them as a Top-1 matching pair. Therefore, the *Precision*@1 of bin_1 vs. bin_2 is calculated as the number of Top-1 matching functions in bin_1 divided by $|B|$ (the number of functions inside bin_1).

Note that *Precision@K* counts functions with the same similarity score. For example, if the first two entries in the resulting function list share the same score, then they are both counted in *Precision@1*. Namely, it is considered to be correct when the target function matches either of the entries. Thus, we further adopt *Single Match Rate* (R_S) to measure the distinguishability of ARCTURUS. It is the rate of correct matches in *Precision@1*, each of which has a single entry for the highest similarity score:

$$R_S = \frac{|\{\, b \in B \mid n_b \in N_L^1 \wedge |N_L^1| = 1 \,\}|}{|\{\, b \in B \mid n_b \in N_L^1 \,\}|},$$

where N_L^1 is the symbol-name set of entries with the highest similarity score.

Baselines. We compare ARCTURUS with eight state-of-the-art baseline techniques that represent four different kinds of similarity analysis approaches:

- **BinDiff** [3] and **ISRD** [86]. These two methods are state-of-the-art syntax-based solutions that measure similarity based on program CFGs or instruction sequences.
- **IMF-Sim** [80] and **CACompare** [40]. These two methods measure the similarity of two functions by directly executing them with identical inputs and comparing outputs.
- **BinGo** [17] and **BLEX** [28]. These two methods are state-of-the-art dynamic similarity analysis techniques that achieve complete code coverage by overriding the intended program logic. Since BLEX is publicly unavailable, we re-implement it based on the same LLVM IR used by ARCTURUS to perform a fair comparison.
- **jTrans** [79], **Asm2Vec** [26], and **SAFE** [58]. These three methods are state-of-the-art deep learning-based solutions for similarity analysis. We implement SAFE [8] and jTrans [9] based on their official PyTorch code. We directly use the analysis platform provided by Asm2Vec [6]. We use their default parameter settings in the evaluation.

For baseline approaches that are publicly available (BinDiff, jTrans, Asm2Vec, SAFE) or are re-implemented (BLEX) by us, we use them to analyze all binaries in our benchmark and compare the results. Since unavailable baselines (ISRD, IMF-Sim,[1] CACompare, BinGo) have the same

[1] IMF-Sim has a Github version, but we are not sure about its validity, so we say that IMF-Sim is closed-source.

Table 3. *Precision@1* Results of Cross-compilation-setting Analysis

Project	G_7^3				G_4^3				C_8^3				C_3^3				G_7^B
	G_7^0	G_4^0	C_8^0	C_3^0	G_7^0	G_4^0	C_8^0	C_3^0	G_7^0	G_4^0	C_8^0	C_3^0	G_7^0	G_4^0	C_8^0	C_3^0	G_7^0
ImageMagick	.870	.868	.871	.869	.894	.897	.890	.891	.925	.924	.937	.936	.946	.943	.956	.956	-
binutils	.869	.870	.859	.861	.876	.883	.854	.854	.903	.907	.917	.915	.913	.917	.926	.928	-
busybox	.885	.885	.882	.882	.891	.886	.894	.893	.899	.897	.901	.900	.903	.899	.907	.906	-
coreutils	.868	.868	.855	.850	.874	.880	.869	.861	.882	.893	.885	.877	.912	.919	.907	.900	.831
diffutils	.861	.868	.854	.847	.831	.834	.831	.828	.891	.895	.877	.873	.931	.935	.913	.913	-
gpac	.835	.835	.832	.831	.846	.846	.840	.839	.879	.884	.871	.873	.883	.883	.878	.880	-
htslib	.920	.919	.910	.910	.913	.916	.909	.908	.936	.939	.941	.942	.944	.949	.951	.951	-
libarchive	.863	.870	.869	.872	.890	.902	.882	.885	.899	.909	.896	.895	.914	.925	.918	.920	-
libxml2	.815	.813	.806	.809	.804	.800	.792	.795	.852	.851	.863	.866	.880	.878	.889	.890	-
nginx	.892	.887	.878	.877	.891	.889	.875	.872	.883	.882	.894	.896	.906	.903	.927	.924	-
openssl	.849	.853	.854	.855	.879	.894	.892	.891	.881	.893	.894	.894	.912	.926	.927	.926	.820
putty	.854	.853	.845	.846	.871	.872	.855	.850	.878	.879	.870	.865	.893	.892	.878	.888	-
sqlite	.834	.833	.817	.817	.806	.811	.793	.794	.868	.882	.875	.873	.886	.901	.886	.886	-
tcpdump	.804	.797	.794	.792	.841	.835	.821	.816	.881	.885	.882	.883	.887	.888	.905	.902	-
transmission	.892	.891	.888	.887	.884	.889	.879	.879	.917	.917	.923	.921	.924	.926	.928	.928	-
vim	.766	.764	.774	.774	.802	.802	.800	.798	.844	.846	.848	.846	.871	.874	.872	.870	-
Average	**.855**	**.855**	**.849**	**.849**	**.862**	**.865**	**.855**	**.853**	**.889**	**.893**	**.892**	**.891**	**.907**	**.910**	**.910**	**.910**	**.826**

L_V^n represents the compilation configuration, meaning that binaries are generated with the compiler L of the version V with optimization option -0n. -0B is the optimization organized by BinTuner [66]. G_7, G_4 is short for GCC v7.5.0, GCC v4.8.1, and C_8, C_3 means Clang v8.0.0, Clang v3.6.2.

projects (e.g., `Coreutils`) in the benchmark as ARCTURUS's and use the same metric to measure the correctness (Equitation 2), we directly refer to the corresponding results in their papers for comparison.

7 EVALUATION

In this section, we design a group of experiments to evaluate the effectiveness and capability of ARCTURUS by investigating the following research questions:

- **RQ1**: How accurate and efficient is ARCTURUS in matching similar binary functions across code transformations?
- **RQ2**: How does ARCTURUS compare to the state-of-the-art?
- **RQ3**: How effective is ARCTURUS in distinguishing similar functions?
- **RQ4**: How effective is ARCTURUS in real-world applications?

7.1 RQ1: Accuracy and Efficiency

In this section, we evaluate the accuracy and efficiency of ARCTURUS in matching similar binary functions across code transformations.

7.1.1 Accuracy. Table 3 lists the *Precision@1* results of -O3 vs. -O0 binaries, and -0B vs. -O0 binaries, which are shown to be the most challenging similarity analysis tasks [66, 79]. BinTuner [66] aims at achieving a higher optimization level than the regular -O3 level, so we regard it as a special case of -O3 optimized code and only compare it with -O0 optimized binaries. Complete *Precision@K* results are presented in Table 4. ARCTURUS achieves an average *Precision@1* of 87.8% for comparison between -O0 and -O3 binaries. The average *Precision@1*, *Precision@3*, and *Precision@5* for all compilation settings are 89.6%, 93.9%, and 95.1%, respectively. It even bridges the syntax and structure gaps created by BinTuner, and the results are comparable to those of conventional settings. Taking `Coreutils` as an example, the *Precision@1* of BinTuner vs. -O0 is 83.1%, while that of -O3 vs. -O0 is 86.8%. Additionally, ARCTURUS achieves 100% code coverage

Table 4. Complete Results of Cross-compilation-setting Analysis

			ImageMagick	binutils	busybox	coreutils	diffutils	gpac	htslib	libarchive	libxml2	nginx	openssl	putty	sqlite	tcpdump	transmission	vim
G_7^3	G_7^0	@1	.870	.869	.885	.868	.861	.835	.920	.863	.815	.892	.849	.854	.834	.804	.892	.766
		@3	.906	.922	.932	.921	.915	.903	.969	.923	.889	.941	.905	.920	.894	.891	.943	.838
		@5	.918	.938	.947	.934	.932	.918	.978	.943	.913	.955	.919	.944	.915	.919	.953	.854
	G_4^0	@1	.868	.870	.885	.868	.868	.835	.919	.870	.813	.887	.853	.853	.833	.797	.891	.764
		@3	.906	.921	.930	.922	.915	.903	.972	.924	.891	.942	.903	.917	.894	.882	.941	.836
		@5	.918	.935	.943	.935	.940	.917	.981	.947	.914	.955	.918	.941	.911	.911	.952	.852
	C_8^0	@1	.871	.859	.882	.855	.854	.832	.910	.869	.806	.878	.854	.845	.817	.794	.888	.774
		@3	.911	.915	.932	.920	.922	.902	.972	.928	.878	.927	.909	.917	.876	.880	.942	.847
		@5	.923	.933	.944	.934	.936	.916	.983	.951	.901	.937	.924	.938	.894	.906	.952	.862
	C_3^0	@1	.869	.861	.882	.850	.847	.831	.910	.872	.809	.877	.855	.846	.817	.792	.887	.774
		@3	.910	.916	.931	.917	.918	.901	.971	.928	.878	.926	.908	.909	.876	.883	.942	.846
		@5	.923	.935	.944	.932	.932	.917	.983	.951	.903	.933	.924	.925	.896	.905	.952	.861
	G_7^2	@1	.917	.906	.937	.868	.907	.915	.960	.911	.851	.954	.893	.903	.922	.917	.941	.848
		@3	.936	.937	.963	.907	.961	.942	.990	.951	.913	.973	.935	.946	.947	.949	.975	.886
		@5	.946	.941	.971	.914	.961	.950	.991	.960	.922	.978	.943	.956	.958	.957	.983	.894
	G_4^2	@1	.895	.887	.922	.903	.877	.894	.949	.882	.833	.937	.874	.884	.905	.893	.923	.799
		@3	.922	.921	.955	.948	.942	.934	.984	.933	.895	.965	.925	.934	.938	.932	.962	.862
		@5	.931	.932	.964	.955	.957	.944	.988	.948	.906	.970	.936	.946	.950	.942	.971	.877
	C_8^2	@1	.893	.902	.913	.911	.872	.869	.926	.918	.821	.905	.883	.882	.862	.880	.942	.813
		@3	.921	.942	.951	.954	.936	.921	.980	.965	.904	.945	.923	.948	.918	.924	.973	.871
		@5	.932	.955	.960	.965	.955	.932	.986	.974	.927	.956	.935	.966	.937	.949	.978	.890
	C_3^2	@1	.883	.894	.912	.913	.896	.881	.923	.914	.823	.902	.880	.885	.859	.884	.943	.812
		@3	.918	.942	.949	.958	.966	.923	.973	.955	.906	.949	.917	.946	.918	.917	.973	.866
		@5	.929	.950	.963	.970	.981	.935	.978	.967	.921	.957	.931	.959	.934	.932	.976	.886
G_4^3	G_7^0	@1	.894	.876	.891	.874	.831	.846	.913	.890	.804	.891	.879	.871	.806	.841	.884	.802
		@3	.929	.922	.937	.929	.905	.901	.957	.941	.889	.957	.925	.929	.876	.908	.939	.856
		@5	.938	.933	.952	.939	.926	.915	.967	.959	.910	.962	.942	.947	.896	.926	.951	.869
	G_4^0	@1	.897	.883	.886	.880	.834	.846	.916	.902	.800	.889	.894	.872	.811	.835	.889	.802
		@3	.928	.925	.933	.930	.912	.901	.960	.948	.890	.957	.938	.931	.877	.899	.941	.853
		@5	.939	.936	.948	.939	.932	.916	.969	.963	.909	.963	.954	.951	.897	.920	.951	.867
	C_8^0	@1	.890	.854	.894	.869	.831	.840	.909	.882	.792	.875	.892	.855	.793	.821	.879	.800
		@3	.922	.918	.938	.926	.912	.901	.957	.941	.872	.939	.934	.926	.858	.885	.938	.854
		@5	.932	.933	.946	.936	.932	.913	.968	.958	.899	.952	.947	.943	.890	.909	.950	.868
	C_3^0	@1	.891	.854	.893	.861	.828	.839	.908	.885	.795	.872	.891	.850	.794	.816	.879	.798
		@3	.923	.918	.939	.916	.909	.900	.957	.941	.875	.940	.933	.917	.858	.883	.938	.851
		@5	.933	.935	.948	.930	.929	.913	.968	.958	.900	.950	.947	.933	.892	.909	.950	.866
	G_7^2	@1	.919	.879	.924	.857	.878	.897	.943	.894	.823	.945	.892	.899	.852	.891	.929	.828
		@3	.940	.919	.958	.905	.932	.929	.976	.946	.896	.975	.936	.936	.917	.937	.964	.876
		@5	.947	.930	.965	.923	.946	.940	.981	.965	.909	.980	.946	.951	.932	.951	.970	.889
	G_4^2	@1	.935	.910	.936	.926	.885	.921	.964	.923	.855	.960	.919	.924	.882	.921	.947	.860
		@3	.953	.934	.961	.956	.936	.947	.987	.960	.904	.979	.956	.950	.926	.949	.976	.894
		@5	.958	.942	.966	.962	.953	.955	.990	.969	.914	.982	.962	.963	.944	.958	.979	.903
	C_8^2	@1	.908	.893	.916	.909	.868	.868	.925	.914	.832	.900	.900	.889	.844	.859	.918	.829
		@3	.935	.943	.951	.952	.952	.916	.972	.963	.919	.940	.937	.938	.902	.904	.957	.871
		@5	.945	.952	.959	.960	.963	.934	.978	.979	.934	.954	.950	.953	.925	.918	.964	.882
	C_3^2	@1	.902	.912	.926	.913	.887	.878	.929	.926	.834	.908	.905	.901	.851	.894	.925	.831
		@3	.932	.942	.966	.954	.949	.922	.970	.964	.917	.949	.940	.949	.911	.926	.965	.874
		@5	.943	.954	.971	.965	.960	.935	.977	.977	.934	.957	.952	.959	.927	.940	.973	.886
C_8^3	G_7^0	@1	.925	.903	.899	.882	.891	.879	.936	.899	.852	.883	.881	.878	.868	.881	.917	.844
		@3	.948	.952	.943	.939	.953	.928	.968	.944	.917	.939	.927	.925	.929	.923	.947	.900
		@5	.962	.964	.954	.955	.978	.944	.976	.961	.936	.951	.936	.945	.949	.934	.958	.912
	G_4^0	@1	.924	.907	.897	.893	.895	.884	.939	.909	.851	.882	.893	.879	.882	.885	.917	.846
		@3	.947	.955	.940	.939	.957	.928	.974	.951	.924	.940	.934	.930	.935	.924	.944	.896
		@5	.961	.969	.951	.953	.982	.942	.981	.962	.939	.955	.945	.948	.948	.936	.955	.910
	C_8^0	@1	.937	.917	.901	.885	.877	.871	.941	.896	.863	.894	.894	.870	.875	.882	.923	.848
		@3	.960	.951	.943	.940	.953	.924	.981	.940	.933	.945	.935	.927	.930	.923	.955	.897
		@5	.972	.966	.954	.955	.967	.938	.985	.953	.952	.956	.946	.948	.938	.933	.963	.913
	C_3^0	@1	.936	.915	.900	.877	.873	.873	.942	.895	.866	.896	.894	.865	.873	.883	.921	.846
		@3	.960	.950	.941	.936	.953	.925	.981	.941	.934	.943	.934	.920	.931	.924	.957	.896
		@5	.972	.965	.954	.950	.967	.939	.985	.953	.954	.956	.945	.938	.939	.934	.965	.914
	G_7^2	@1	.929	.905	.916	.858	.893	.893	.936	.927	.873	.915	.905	.895	.886	.947	.943	.868
		@3	.960	.951	.951	.918	.963	.933	.976	.957	.934	.961	.943	.943	.951	.967	.968	.917
		@5	.976	.962	.961	.932	.974	.945	.981	.963	.945	.969	.949	.952	.965	.975	.975	.930
	G_4^2	@1	.923	.908	.912	.914	.892	.900	.930	.917	.852	.912	.908	.877	.877	.918	.938	.858
		@3	.953	.950	.943	.955	.959	.942	.972	.956	.924	.956	.946	.939	.937	.946	.962	.903
		@5	.964	.955	.954	.966	.978	.954	.986	.965	.942	.965	.954	.950	.953	.960	.967	.919
	C_8^2	@1	.989	.994	.991	.986	1.000	.981	.993	.991	.973	.980	.988	.992	.979	.971	.992	.971
		@3	.992	.997	.994	.997	1.000	.990	.998	.998	.988	.986	.993	.997	.988	.978	.998	.979
		@5	.997	.997	.995	.997	1.000	.992	.998	.998	.991	.993	.994	.998	.993	.982	.998	.981
	C_3^2	@1	.967	.974	.972	.964	.949	.959	.971	.964	.936	.962	.962	.947	.945	.943	.981	.936
		@3	.981	.989	.987	.981	.996	.977	.987	.986	.978	.981	.977	.972	.980	.956	.992	.959
		@5	.988	.990	.988	.983	.996	.982	.988	.990	.985	.990	.980	.977	.988	.965	.995	.963
	G_7^0	@1	.946	.913	.903	.912	.931	.883	.944	.914	.880	.906	.912	.893	.886	.887	.924	.871
		@3	.964	.957	.945	.958	.953	.943	.973	.953	.929	.957	.948	.928	.940	.921	.957	.922
		@5	.965	.967	.954	.967	.964	.955	.981	.965	.946	.963	.956	.954	.954	.930	.967	.931
	G_4^0	@1	.943	.917	.899	.919	.935	.883	.949	.925	.878	.903	.926	.892	.901	.888	.926	.874
		@3	.962	.962	.942	.955	.957	.944	.977	.959	.937	.956	.957	.926	.947	.922	.954	.920
		@5	.964	.971	.953	.966	.968	.954	.985	.970	.949	.966	.964	.951	.954	.933	.963	.929

(Continued)

Table 4. Continued

			ImageMagick	binutils	busybox	coreutils	diffutils	gpac	htslib	libarchive	libxml2	nginx	openssl	putty	sqlite	tcpdump	transmission	vim
C_3^3	C_8^0	@1	.956	.926	.907	.907	.913	.878	.951	.918	.889	.927	.927	.878	.886	.905	.928	.872
		@3	.972	.957	.946	.957	.953	.936	.981	.962	.942	.964	.958	.921	.933	.943	.961	.918
		@5	.973	.967	.955	.966	.964	.949	.987	.973	.959	.969	.965	.945	.940	.948	.968	.931
	C_3^0	@1	.956	.928	.906	.900	.913	.880	.951	.920	.890	.924	.926	.888	.886	.902	.928	.870
		@3	.972	.957	.945	.955	.949	.939	.983	.962	.942	.962	.957	.933	.933	.939	.962	.917
		@5	.975	.970	.955	.965	.960	.951	.987	.974	.959	.969	.965	.953	.940	.945	.970	.930
	G_7^2	@1	.931	.910	.934	.879	.941	.883	.941	.923	.886	.935	.910	.894	.893	.900	.938	.883
		@3	.967	.949	.962	.930	.963	.924	.973	.959	.941	.970	.948	.933	.954	.931	.968	.931
		@5	.972	.959	.968	.939	.967	.937	.982	.968	.952	.975	.956	.944	.968	.943	.976	.942
	G_4^2	@1	.932	.924	.927	.924	.897	.899	.939	.927	.882	.936	.923	.884	.908	.902	.939	.883
		@3	.961	.957	.953	.967	.952	.943	.967	.956	.937	.967	.955	.937	.951	.933	.965	.924
		@5	.965	.961	.960	.976	.974	.955	.982	.964	.952	.970	.963	.952	.964	.947	.971	.938
	C_8^2	@1	.983	.983	.973	.970	.953	.951	.976	.974	.968	.968	.963	.938	.968	.928	.982	.973
		@3	.992	.993	.986	.983	.993	.967	.987	.986	.989	.982	.982	.966	.984	.945	.993	.984
		@5	.993	.995	.988	.986	.993	.982	.991	.994	.993	.991	.985	.971	.991	.947	.993	.987
	C_3^2	@1	.997	.993	.994	.995	1.000	.988	.997	.987	.984	.994	.992	.990	.985	.997	.993	.988
		@3	.997	.997	.997	.996	1.000	.993	.999	.993	.994	.997	.996	.997	.994	1.000	.999	.992
		@5	.997	.998	.997	.996	1.000	.994	.999	.995	.996	.997	.996	.997	.995	1.000	.999	.993
G_7^B	G_7^0	@1	-	-	-	.831	-	-	-	-	-	-	.820	-	-	-	-	-
		@3	-	-	-	.881	-	-	-	-	-	-	.860	-	-	-	-	-
		@5	-	-	-	.935	-	-	-	-	-	-	.908	-	-	-	-	-

L_V^n represents the compilation configuration, meaning that binaries are generated with the compiler L of the version V with optimization option -0n. -0B is the optimization organized by BinTuner [66]. G_7, G_4 is short for GCC v7.5.0, GCC v4.8.1, and C_8, C_3 means Clang v8.0.0, Clang v3.6.2. @K means Precision@K.

Table 5. *Precision@1* Results of Matching Functions Obfuscated by OLLVM

Project	OLLVM -O3 vs. GCC/Clang -O0			
	BCF	FLA	SUB	ALL
coreutils	.860	.799	.887	.705
diffutils	.875	.749	.890	.667
htslib	.938	.781	.947	.674
libarchive	.891	.769	.902	.683
nginx	.838	.709	.890	.630
openssl	.791	.632	.898	.608
transmission	.885	.727	.921	.666
Average	**.868**	**.738**	**.905**	**.662**

(BCF: Bogus Control Flow, FLA: Control Flow Flattening, SUB: Instructions Substitution, ALL: BCF + FLA + SUB)

in all projects, which demonstrates that ARCTURUS is able to achieve high accuracy and full code coverage simultaneously due to its awareness of execution context consistency.

The results of ARCTURUS to match obfuscated functions with benign ones are presented in Table 5. ARCTURUS does well in handling functions obfuscated by BCF and SUB. The average *Precision@1* is 86.8% and 90.5% separately.

7.1.2 False Case Analysis. We have manually inspected the comparison results to find the root cause of the incorrect matching. After inspecting, we find that the main reason leading to the false cases is the **function inlining and splitting**. Modern compilers tend to inline or split user-defined/library functions to enable further optimizations. These two optimizations break the function boundaries and re-arrange the semantics contained by each function. Specifically, inlining expands a callee into its callers, introducing extra semantics (i.e., those of the callee) to the caller. By contrast, splitting breaks a function into parts and connects them with inter-procedural calls; each fragment only contains a subset of the original semantics. As a result, the features collected by ARCTURUS will change accordingly, and it will miss the match between functions with and

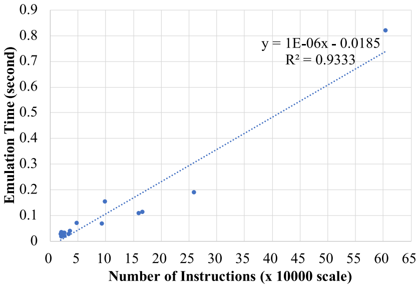


Fig. 5. Linear fit lines on the emulation time. The x-axis stands for the number of instructions of function (*times* 10,000 scale). The y-axis stands for the time cost (second) or the emulation.

Table 6. Average Online Time Cost for Each Stage (in Seconds)

Project	Reachability Calculation	Emulation	Similarity Comparison	Total
ImageMagick	95.81	2035.16	55.47	2186.44
binutils	9.94	50.22	8.24	68.40
busybox	7.82	61.76	16.47	86.04
coreutils	4.60	30.65	3.45	38.70
diffutils	0.82	4.04	0.34	5.21
gpac	85.72	1133.37	304.58	1523.68
htslib	3.42	19.88	2.01	25.31
libarchive	3.26	25.50	4.99	33.75
libxml2	8.92	108.78	23.98	141.68
nginx	1.47	17.22	3.05	21.74
openssl	15.26	446.41	86.36	548.03
putty	7.26	88.88	4.51	100.65
sqlite	24.26	185.58	10.23	220.07
tcpdump	4.63	21.52	3.42	29.57
transmission	8.30	30.97	4.54	43.81
vim	55.97	764.90	108.56	929.44
Average (Percentage)	21.9 (5%)	314.5 (84%)	40.01 (11%)	375.16
Time/Function	**0.01**	**0.13**	**0.02**	**0.15**

without inlining/splitting. Completely solving this problem requires inlining all callee functions into its caller, but this might cause the program to blow up in size exponentially. Selective inlining [17] could also help alleviate the problem, but it is hard to determine the correct threshold such that the inlining degrees between different binaries are the same.

We found that another problem is the **lack of function features**. ARCTURUS will fail to find correct matches for functions without semantic features, especially those that only perform arithmetic operations with few I/O behaviors, e.g., encryption functions in `OpenSSL`. Fortunately, due to the complete code coverage, ARCTURUS is able to capture rich semantics information, and only 0.43% of the functions are found to have no features in the experiments.

For code obfuscation, FLA becomes the obstacle in the way of ARCTURUS attempting to achieve high accuracy, and the average *Precision@1* is only 73.8%. The main reason is that FLA breaks an original path into multiple pieces and chains them via infinite loops at runtime. However, ARCTURUS pursues full block coverage instead of full path coverage, which, unfortunately, is still an open problem [32]. Thus, the resulting semantics captured are far from complete, accounting for only a small part of the original paths, which results in incorrect matching.

7.1.3 *Efficiency.*

We evaluate the efficiency of ARCTURUS by measuring the average online time spent by different steps of the analysis. Table 6 presents the results. Overall, ARCTURUS spends an average time of 375.16 s on one binary, including code reachability analysis, emulation, and similarity comparison. ARCTURUS only spends 0.15 s, on average in handling one function, showing that it is sufficiently efficient for practical use. Reachability analysis only accounts for 5% of the whole processing time, on average. In contrast, due to the requirement of full code coverage, emulation costs the majority of the entire time, i.e., 84%. The time spent on similar function searching (e.g., similarity comparison) is also negligible, accounting for only 11% of the total running time. On average, only 0.02 s is spent on searching for similar functions in the function pool.

We adopt the curve-fitting approach [68] to study the observed average emulation time consumption of one function for each project. Figure 5 shows the fitting curves and their coefficients of determination R^2. $R^2 \in [0, 1]$ is a statistical measure of how close the data are to the fitting curve. The closer R^2 is to 1, the better the fitting curve is. It shows that ARCTURUS's emulation

Table 7. Average *Precision@1* Results and Processing Time of Baselines

	ARCTURUS			BinDiff	ISRD	BinGo	BLEX	BLEX-V	CACompare	IMF-Sim	Asm2Vec	jTrans	SAFE
	*	coreutils	P_{cac}	*	coreutils	coreutils	*		P_{cac}	coreutils	*	*	*
G^3 vs. G^0	.859	.873	.875	.137	.624	.331	.591	.614	.768	.704	.333	.571	.253
G^3 vs. C^0	.852	.859	.872	.121	.644	.326	.573	.584	-	.601	.327	.452	.222
C^3 vs. C^0	.901	.892	.909	.174	.784	.344	.640	.663	-	.775	.353	.487	.290
C^3 vs. G^0	.899	.901	.906	.149	.562	.343	.639	.677	-	.660	.344	.581	.285
Average	**.878**	**.881**	**.890**	.145	.654	.336	.611	.634	.768	.685	.339	.523	.262
Time/Function	0.15	0.03	0.26	**0.01**	-	-	25.18		5.20	11.37	0.17	0.81	0.36

C^n and G^n mean binaries compiled by GCC (v7.5.0/4.8.1) and Clang (v8.0.0/3.6.2) with optimization option `-O`n, respectively. BLEX-V means the **V**ariant of BLEX that adopts the same semantics features as ARCTURUS. $*$ means to take all the projects as the benchmark. P_{cac} means the shared four projects in our benchmark, which are also adopted by CACompare. "-" means such data is not available. The processing time of BLEX and BLEX-V refers to the original implementation based on dynamic instrumentation (25.18 s/function = 57 CPU days / 195,560 functions) [28].

time grows almost linearly with the complexity of the function in practice ($R^2 > 0.93$), thus scaling up quite gracefully.

7.2 RQ2: Baseline Comparison

In this section, we compare ARCTURUS with the state-of-the-art similarity analysis techniques, including the syntax-based method of **BinDiff** [3] and **ISRD** [86]; the dynamic solutions of **BLEX** [28], **BinGo** [17], **IMF-Sim** [80], and **CACompare** [39]; the learning-based approaches of **jTrans** [79], **Asm2Vec** [26], and **SAFE** [58]. We first compare their performance on matching benign code across compilation settings, then leverage them to analyze OLLVM-obfuscated code and compare their results.

7.2.1 Benign Code Analysis. We compare the performance of ARCTURUS with baseline techniques under the most challenging setting, i.e., matching -O0 and -O3 binaries compiled by different compiler versions. For unavailable baselines such as BinGo [17], IMF-Sim [80], and CACompare [40], we show the results of running ARCTURUS on datasets in their papers and directly compare the results with their reported *Precision@1*. Additionally, to conduct a fair comparison, we also re-implement the variant of BLEX [28] to adopt the same semantics features as ARCTURUS, denoted as **BLEX-V**.

The results are presented in Table 7. The *Precision@1* result of ARCTURUS outperforms all baseline approaches by 38.4%, on average. ARCTURUS also runs faster than all compared tools, except BinDiff. Syntax-based solutions such as BinDiff and ISRD have worse performance than ARCTURUS, because they rely on the CFG structures or instruction sequences to compare binaries, which could be notably altered by heavy compiler optimizations. Dynamic similarity analysis methods (IMF-Sim and CACompare) that are based on native executions have better *Precision@1* than syntax-based solutions, because they both leverage input/output behaviors during the actual executions, which are more resilient towards code transformations. However, both of them have worse *Precision@1* results than that of ARCTURUS, because they both adopt comparison information and memory variable offsets as code features, which are not robust enough to code transformations. Moreover, these two methods suffer from the coverage issue, which limits their distinguishability (see Section 7.3). Although IMF-Sim leverages advances in fuzzing techniques to increase the code coverage, it only covers 31.8% of the instructions [80].

Although three existing solutions (i.e., BLEX, BLEX-V, and BinGo) can also achieve high code coverage as ARCTURUS, they do not perform as well as ARCTURUS in terms of the *Precision@1* results (on average, 35.2% worse). The root cause is that they all disregard the context consistency, i.e., semantically equivalent functions might be executed with different program states, thus result-

Table 8. Comparison Results with Baselines on Obfuscated Code

		ARCTURUS		BLEX	BLEX-V	IMF-Sim	Asm2Vec
		*	coreutils	*		coreutils	*
OLLVM -O3 vs. GCC/Clang -O0	BCF	.868	.860	.608	.661	.449	.331
	FLA	.738	.799	.620	.655	.613	.242
	SUB	.905	.887	.663	.669	.722	.334
	ALL	.662	.705	.560	.616	-	.202
Average		**.793**	**.813**	.612	.650	.594	.277

(BCF: Bogus Control Flow, FLA: Control Flow Flattening, SUB: Instructions Substitution, ALL: BCF + FLA + SUB) BLEX-V means the **V**ariant of BLEX that adopts the same semantics features as ARCTURUS. * means to take all the projects as the benchmark. "-" means such data is not available.

ing in mismatches. BLEX performs not so well as BLEX-V in terms of analysis accuracy, because it adopts stack variable values as features, which are unstable when faced with code transformations.

Three learning-based methods (i.e., Asm2Vec, jTrans, and SAFE) also perform poorly on our dataset. All of them infer the representation (i.e., an embedding) of binary code by capturing sequences of instructions and CFGs [26, 58, 79], which still heavily depend on both the compiler versions and optimization levels used. The provided models are usually trained on specific datasets, which are also generated from given compiler versions. Therefore, these approaches might work on existing datasets but fail on new or unseen datasets.

7.2.2 Efficiency Comparison. Table 7 also presents the average processing time for one function. ARCTURUS takes only 0.15 second to process one function, on average, which is much more efficient than all the other dynamic solutions. Existing dynamic methods such as BLEX and IMF-Sim are implemented based on Pin, which brings much overhead during execution [77]. On the contrary, ARCTURUS's emulation engine is based on the interpreter of the LLVM IR, which avoids expensive system-level operations and takes much less time to handle one function. BinDiff takes less time than ARCTURUS because it only performs simple graph-isomorphism checking on functions' CFGs, leading to very low accuracy. ARCTURUS is even more scalable than all learning-based methods, such as Asm2Vec, showing that it is practical enough for real-world usage.

7.2.3 Obfuscated Code Analysis. We also compare ARCTURUS with the baselines to match obfuscated binaries. The experimental results are presented in Table 8. Generally, since OLLVM is based on Clang, all the methods perform worse than analyzing benign code with the configuration of Clang -O3 vs. GCC/Clang -O0 (Table 7). Despite that, ARCTURUS still outperforms the others by 26.5%, on average.

7.3 RQ3: Distinguishability

In this section, we study the distinguishability of ARCTURUS and compare its capability with other solutions. We create an ablation of ARCTURUS, named ARCTURUS-S, which only emulates one path normally without overriding any branch outcome, similar to existing dynamic-based methods like IMF-Sim.

Figure 6 shows the results of the cross-compilation-setting analysis (-O0 and -O3). The reasons that lead to incorrect matching of ARCTURUS, including *function inlining*, have fewer effects on single-path emulation. Thus, the *Precision@1* of ARCTURUS is slightly lower than that of ARCTURUS-S (87.8% vs. 91.2%). However, the single match rate of the former is much better than that of the latter (73.5% vs. 57.4%). The above results confirm our arguments about the limitations of the native execution in Section 3.1. The native execution could hardly distinguish programs that

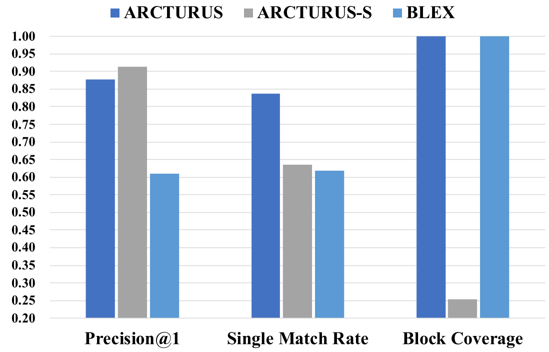


Fig. 6. Results of distinguishability study. ARCTURUS-S only performs native execution.

have similar shallow paths but with different deeply embedded behaviors. Therefore, it could draw the wrong conclusion that two binary functions are from the same source code and mistakenly rank many unmatched functions as Top-1, resulting in a high *Precision@1* but a low single-match rate.

Note that the block coverage of ARCTURUS-S is only 25.3%. The results also indicate that the similar solutions of ARCTURUS-S, which has limited coverage (e.g., CACompare [40] and IMF-Sim [80]), would suffer from the distinguishability issue as well. Despite the complete block coverage, BLEX still produces a low single match rate, because it executes similar code fragments of dissimilar functions with the same initial program state, making them indistinguishable from each other. Compared to ARCTURUS-S, BLEX-V has a much lower *Precision@1* but a comparable single match rate, showing that the distinguishability of similarity analysis is indeed improved by covering more code to enrich semantics information.

The performance of ARCTURUS in distinguishability highlights its usefulness in practice. For a target function, if there are multiple entries with the same highest score in the resulting reference function list, then human analysts will have to examine each entry to find the correct one, which is tedious and prone to errors.

7.4 RQ4: Applications

In this section, we show the effectiveness of ARCTURUS in real-world scenarios, including known vulnerability detection and binary version identification.

7.4.1 Known Vulnerability Detection. We choose five different kinds of CVEs in the `OpenSSL` project for the evaluation, including the famous Heartbleed (CVE-2014-0160) vulnerability, buffer overflow (CVE-2014-0195), and so on. We compile seven applications (`curl`, `libmariadb`, `nginx`, `wget`, `links`, `git`, `exim`) with the statically linked `OpenSSL` library using four different compiler versions (GCC v7.5.0/4.8.1 and Clang v8.0.0/3.6.2) with the projects' default compilation configurations. For each CVE, we use the vulnerable binary function as the single query target and search it against all functions inside each application binary. Figure 7 presents the average *Precision@1*, single match rate, and *Precision@3* of all seven applications. The average *Precision@1* and single match rate are all above 92%, and the *Precision@3* even reaches 100%. Moreover, the processing time of each program is no more than 4 minutes, indicating ARCTURUS's capability in practical applications.

7.4.2 Binary Version Identification. Binary version detection is an essential step in knowledge reuse. With the reference code, ARCTURUS can determine the correct version of unknown bi-

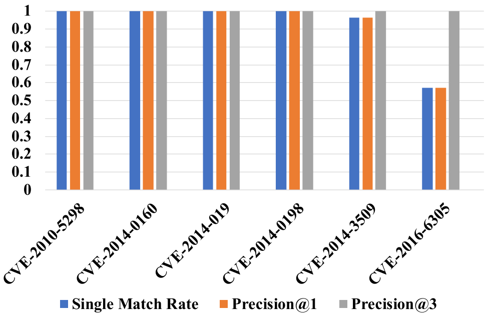


Fig. 7. Known vulnerability detection results on OpenSSL.

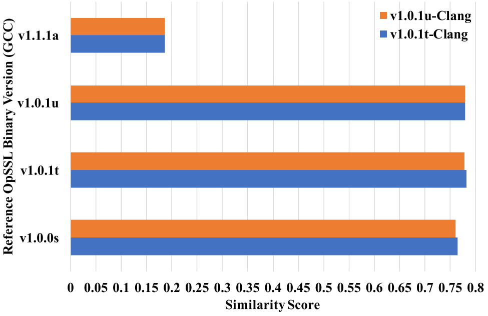


Fig. 8. Results of binary version identification on OpenSSL.

nary programs via similarity analysis. Specifically, similar to CoP [56], ARCTURUS measures the likeness of two binary files with the average highest similarity scores of all matched function pairs.

We adopt four different versions of OpenSSL in the experiment: v1.1.1a, v1.0.1u, v1.0.1t, and v1.0.0s. All four versions of the source code are compiled into binaries with GCC and are used as the reference code. We use ARCTURUS to identify the versions of two Clang-compiled unknown OpenSSL binaries B_u and B_t, whose real versions are v1.0.1u and v1.0.1t, respectively.

Figure 8 presents the results of comparing the two target binaries with four reference OpenSSL binaries. For the target binary B_u whose correct version is v1.0.1u, the likeness scores between it and four reference binaries (v1.1.1a, v1.0.1u, v1.0.1t, and v1.0.0s) are **0.186**, **0.780**, **0.778**, and **0.761**, respectively. ARCTURUS produces a higher similarity score for the correct matched pair (v1.0.1u vs. v1.0.1u) than all the others. Similarly, ARCTURUS also successfully identifies the correct version of B_t as v1.0.1t by producing a score of **0.782**.

Moreover, we could also infer that there is a larger variance between v1.1.1a and v1.0.1u than between v1.0.1u and v1.0.0s, which is in accordance with their version numbers. The results also show that ARCTURUS is capable of giving high confidence for correctly matched pairs while giving low scores for wrongly matched pairs, which is useful in real-world scenarios where a concrete threshold needs to be established for dealing with a potentially large number of candidate functions.

8 DISCUSSION

Limitations and Future Work. Currently, the implementation of ARCTURUS only handles ELF files on the x64 architecture. Because of LLVM-IR, it could be adapted to other architectures easily (e.g., ARM, MIPS) and file formats (e.g., PE and Mach-O). Even though obfuscation indeed poses difficulties for ARCTURUS (Table 5), it still produces much more dependable results than the state-of-the-art techniques (Table 8). Actually, ARCTURUS is designed for similarity analysis instead of de-obfuscation. Since de-obfuscation has been well-studied [76, 83, 89, 90], it is recommended to first de-obfuscate the binary, which has been obfuscated, then apply ARCTURUS for better results. It is still an open question for binary similarity analysis to detect inlined/split functions. The heuristic solution of selective inlining [17] cannot be applied to general cases. Since the original function boundaries are removed, it is difficult to recover them via reverse engineering. In contrast, re-optimization [20] might be a solution to narrow the gaps caused by such optimizations.

Threats to Validity. ARCTURUS performs reachability-guided emulation on the lifted LLVM-IR. However, binary disassembling/lifting is still an open problem, and the results might contain errors [12, 18, 62]. Incorrect IR might result in dissimilar semantic features. In addition, the current lifter cannot cover all the instructions of x64 [51]. Unmodeled instruction sets, e.g., streaming SIMD extensions, will cause semantic discrepancy. Notwithstanding, it is capable of handling most cases in practice [11], as indicated by the experimental results.

Solutions to Lack of Function Features. Currently, ARCTURUS adopts I/O values as semantics features (Section 4.2). For functions lacking such features, a possible solution is to use dependable structural ones as complements, such as strings [73], argument lists [39], and so on.

Scalability. Emulation is the most time-consuming step of ARCTURUS (Table 6). Fortunately, the process of each function is independent of each other. Thus, it can be embarrassingly parallelized, allowing ARCTURUS to be applied to even larger projects in the real world.

Malware Analysis. Currently, ARCTURUS still cannot handle malware with data stream obfuscation such as VMProtect [1] and self-modifying code [57]. This is because the main challenge of handling such cases lies in binary lifting instead of similarity analysis. We leave supporting malware analysis as one of our future directions.

9 RELATED WORK

Binary Similarity Analysis. Existing binary similarity analysis methods can be categorized as syntax-based and semantics-based. Syntax-based methods leverage syntactic and structural features. BinDiff [3], BinSlayer [14], and discovRE [29] compute the similarity between functions based on the structure of the control-flow graphs. GitZ [20] and ImOpt [43] find strands equality through re-optimization. FirmUp [21] leverages game theory to detect common vulnerabilities in firmware. TRACY [22] BinXray [87], PDiff [45], ISRD [86], and VIVA [82] decompose CFGs into partial traces and compute the edit distance to measure the similarity. Recently, several methods leverage advances in deep learning to compare similarities. Genius [31], VulSeeker [34], and Gemini [85] build a graph embedding for the ACFG of a function, i.e., a CFG with nodes annotated with selected basic block features. αDiff [54] uses a Siamese network with CNN to generate function embeddings. InnerEye [100], SAFE [58], Asm2Vec [26], DeepBinDiff [27], Trex [63], Codee [92], Asteria [93], PalmTree [53], and jTrans [79] utilize various deep learning-based embeddings to capture the information and dependencies of instructions automatically.

Semantics-based methods measure the similarity of binaries by analyzing their functionalities or runtime behaviors. BinHunt [33, 59], CoP [56], Xmatch [30], Esh [19], BinSim [60], and FIBER [96] extract symbolic formulas to check semantic equivalence between binaries. MockingBird [39], CACompare [40], BinMatch [41], and IMF-Sim [80] capture the runtime behaviors of binaries via executing the code with predefined inputs. However, they still suffer from low code coverage, which results in their bad performance in distinguishing between different functions. Multi-MH [65] and BinGo [17, 88] break the code into smaller fragments and extract semantic information from sampling basic blocks. The semantic information is extracted from partial traces without local execution context, which is usually meaningless and would cause incorrect matching. BLEX [28] executes the target function repeatedly, starting from so-far uncovered instructions until every instruction is executed at least once.

Forced Execution. Forced execution is a good choice to extract the semantics of programs with high code coverage dynamically. Essentially, it trades analysis accuracy for code coverage by breaking the normal program flows such that the execution can be started at any program point. Zhang et al. [97] propose locating faults by force-switching conditional branch outcomes. Limbo [81], Micro-Execution [35], X-Force [64], PMP [94], Johnson et al. [47], and Dual-Force [75] force-executed the target program to expose its runtime behaviors. FXE [84] and iRiS [25] construct program control-flow graphs from forced execution results. J-Force [50], JSForce [38], and PMForce [71] apply forced execution to web security.

10 CONCLUSION

We develop a full coverage binary similarity analysis framework that features a novel reachability-guided emulation technique. It directly covers each code block by dynamically forcing a set of branch outcomes under the guidance of code reachability relations. The runtime behaviors are used for similarity comparison. We prove that the executed paths for semantically equivalent binaries produce the same execution results. The experimental results show that it is substantially more effective and efficient than the state-of-the-art approaches.

REFERENCES

[1] VMPSofe. 2017. VMPROTECT SOFTWARE. [Online]. Available: http://vmpsoft.com/

[2] GrammaTech. 2019. Binary Software Composition Analysis. [Online]. Available: https://www.verifysoft.com/en_grammatech_codesentry.html/

[3] Zynamics. 2020. BinDiff. [Online]. Available: https://www.zynamics.com/bindiff/manual/index.html

[4] 2020. BinTuner. Retrieved from https://github.com/BinTuner/Dev

[5] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. *The BSD Conference* 5 (2008), 1–20.

[6] 2020. The Kam1n0 Assembly Analysis Platform. Retrieved from https://github.com/McGill-DMaS/Kam1n0-Community

[7] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. *IEEE/ACM 1st International Workshop on Software Protection*, IEEE, 3–9.

[8] 2021. SAFEtorch. Retrieved from https://github.com/facebookresearch/SAFEtorch

[9] 2022. jTrans. Retrieved from https://github.com/vul337/jTrans

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, & Tools*. Pearson Education India.

[11] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. X86-64 instruction usage among C/C++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 68–79.

[12] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association.

[13] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'09)*. Citeseer, 8–11.

[14] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 1–10.

[15] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'08)*. IEEE.

[16] Nguyen Anh Quynh. 2020. Capstone. *The Ultimate Disassembler*. [Online]. Available: https://www.capstone-engine.org/

[17] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM.

[18] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–671.

[19] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM.

[20] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*.

[21] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM.

[22] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM.

[23] Jack W. Davidson and Sanjay Jinturkar. 1995. *An Aggressive Approach to Loop Unrolling*. Technical Report. Citeseer.

[24] Rocco De Nicola. 2011. Behavioral equivalences. *Encyclopedia of Parallel Computing*, Springer, 120–127.

[25] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iRiS: Vetting private API abuse in iOS applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 44–56.

[26] S. H. Ding, B. C. Fung, and P. Charland. 2019. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19).* IEEE.

[27] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning program-wide code representations for binary diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20).*

[28] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium (SEC'14).* USENIX Association.

[29] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16).*

[30] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17).* ACM.

[31] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'16).* ACM.

[32] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18).* IEEE, 679–696.

[33] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security.* Springer, 238–255.

[34] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18).* IEEE, 896–899.

[35] Patrice Godefroid. 2014. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering.* 539–549.

[36] David Molnar, P. Godefroid, and M. Y. Levin. 2008. Automated whitebox fuzz testing. *NDSS*, Vol. 8, 151–166.

[37] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM Comput. Surv.* 54, 3 (2021), 1–38.

[38] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2017. JSForce: A forced execution engine for malicious JavaScript detection. In *Proceedings of the International Conference on Security and Privacy in Communication Systems.* Springer, 704–720.

[39] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16).* IEEE.

[40] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17).* IEEE.

[41] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A semantics-based hybrid approach on binary code clone analysis. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18).* IEEE.

[42] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13).* 81–96.

[43] Jianguo Jiang, Gengwang Li, Min Yu, Gang Li, Chao Liu, Zhiqiang Lv, Bin Lv, and Weiqing Huang. 2020. Similarity of binaries across optimization levels and obfuscation. In *Proceedings of the European Symposium on Research in Computer Security.* Springer, 295–315.

[44] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 1383–1395.

[45] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.* 1149–1163.

[46] Donald B. Johnson. 1973. A note on Dijkstra's shortest path algorithm. *J. ACM* 20, 3 (1973), 385–388.

[47] Ryan Johnson and Angelos Stavrou. 2013. Forced-path execution for android applications on x86 platforms. In *Proceedings of the IEEE 7th International Conference on Software Security and Reliability Companion*. IEEE, 188–197.

[48] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'21)*.

[49] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *arXiv preprint arXiv:2011.10749* (2020).

[50] Kyungtae Kim, I. Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web*. 897–906.

[51] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE Press.

[52] Liyi Li and Elsa L. Gunter. 2020. K-LLVM: A relatively complete semantics of LLVM IR. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP'20)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[53] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an assembly language model for instruction embedding. *arXiv preprint arXiv:2103.03809* (2021).

[54] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY.

[55] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM.

[56] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM.

[57] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE, 431–441.

[58] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-attentive function embeddings for binary similarity. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329.

[59] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 92–109.

[60] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium (SEC'17)*. USENIX Association.

[61] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized semantics-based binary diffing with application to malware lineage inference. In *Proceedings of the IFIP International Information Security and Privacy Conference*. Springer, 416–430.

[62] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'21)*. IEEE Computer Society, 833–851.

[63] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[64] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: Force-executing binary programs for security applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. 829–844.

[65] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE.

[66] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[67] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the International Symposium on Software Testing and Analysis*. 100–110.

[68] L. A. Sandra. 1994. *PHB Practical Handbook of Curve Fitting*. CRC Press, Boca Raton, FL.

[69] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th International Conference on Supercomputing*. 153–166.

[70] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry practice of coverage-guided enterprise Linux kernel fuzzing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 986–995.

[71] Marius Steffens and Ben Stock. 2020. PMForce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 493–505.

[72] Wei Tang, Du Chen, and Ping Luo. 2018. BCFinder: A lightweight and platform-independent tool to find third-party components in binaries. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC'18)*. IEEE, 288–297.

[73] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A cross-platform and accurate system to detect third-party libraries in binary code. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. IEEE, 104–115.

[74] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 423–434.

[75] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-Force: Understanding webview malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 714–725.

[76] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. 2005. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE.

[77] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. 2007. Analyzing dynamic binary instrumentation overhead. *Workshop on Binary Instrumentation and Application.*

[78] Jeffrey D. Ullman. 1973. Fast algorithms for the elimination of common subexpressions. *Act. Inform.* 2 (1973), 191–213.

[79] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-aware transformer for binary code similarity. *arXiv preprint arXiv:2205.12713* (2022).

[80] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE.

[81] Jeffrey Wilhelm and Tzi-cker Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 219–235.

[82] Yang Xiao, Zhengzi Xu, Weiwei Zhang, Chendong Yu, Longquan Liu, Wei Zou, Zimu Yuan, Yang Liu, Aihua Piao, and Wei Huo. 2021. VIVA: Binary level vulnerability identification via partial signature. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*. IEEE, 213–224.

[83] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. ACM.

[84] Liang Xu, Fangqi Sun, and Zhendong Su. 2009. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep.*, Citeseer, 14–23.

[85] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, New York, NY.

[86] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 873–884.

[87] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387.

[88] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-OS binary code search with emulation. *IEEE Trans. Softw. Eng.* 45, 11 (2018), 1125–1149.

[89] Babak Yadegari and Saumya Debray. 2015. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM.

[90] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE.

[91] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: Binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405.

[92] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244.

[93] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. 2021. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'21)*. IEEE, 224–236.

[94] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective forced execution with probabilistic memory pre-planning. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'20)*. IEEE, 1121–1138.

[95] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Piao Aihua, Xue Jingling, and Huo Wei. 2019. B2SFinder: Detecting open-source software reuse in COTS software. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, IEEE, 1038–1049.

[96] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *Proceedings of the 27th USENIX Security Symposium (SEC'18)*. USENIX Association.

[97] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*. 272–281.

[98] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 427–440.

[99] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. 2020. PatchScope: Memory object centric patch diffing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 149–165.

[100] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'19)*.

Received 2 September 2023; revised 4 December 2023; accepted 18 December 2023