CodeExtract: Enhancing Binary Code Similarity Detection with Code Extraction Techniques

Lichen Jia
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
lcjia457@gmail.com

Chenggang Wu
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
wucg@ict.ac.cn

Peihua Zhang
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
zhangpeihua@ict.ac.cn

Zhe Wang*
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
wangzhe12@ict.ac.cn

Abstract

In the field of binary code similarity detection (BCSD), when dealing with functions in binary form, the conventional approach is to identify a set of functions that are most similar to the target function. These similar functions often originate from the same source code but may differ due to variations in compilation settings. Such analysis is crucial for applications in the security domain, including vulnerability discovery, malware detection, software plagiarism detection, and patch analysis. Function inlining, an optimization technique employed by compilers, embeds the code of callee functions directly into the caller function. Due to different compilation options (such as O1 and O3) leading to varying levels of function inlining, this results in significant discrepancies between binary functions derived from the same source code under different compilation settings, posing challenges to the accuracy of state-of-the-art (SOTA) learning-based binary code similarity detection (LB-BCSD) methods. In contrast to function inlining, code extraction technology can identify and separate duplicate code within a program, replacing it with corresponding function calls. To overcome the impact of function inlining, this paper introduces a novel approach, CodeExtract. This method initially utilizes code extraction techniques to transform code introduced by function inlining back into function calls. Subsequently, it actively inlines functions that cannot undergo code extraction, effectively eliminating the differences introduced by function inlining. Experimental validation shows that CodeExtract enhances the accuracy of LB-BCSD models by 20% in addressing the challenges posed by function inlining.

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

Keywords: Learning-based Binary Similarity Analysis, Function Inline, Program Analysis

ACM Reference Format:
Lichen Jia, Chenggang Wu, Peihua Zhang, and Zhe Wang. 2024. CodeExtract: Enhancing Binary Code Similarity Detection with Code Extraction Techniques. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652032.3657572

1 Introduction

Binary code similarity detection [2, 4, 5, 15–18, 22, 28, 36, 38] plays a crucial role in security-related applications, including vulnerability discovery [9, 11, 19], malware detection [7, 8, 25, 26, 37, 59], software plagiarism detection [30, 31, 44, 48, 60, 61], and patch analysis[21, 34, 40, 51]. With the rapid development of artificial intelligence technologies, LB-BCSD methods have demonstrated superior ability in recognizing the semantics of instructions, surpassing traditional BCSD methods in both accuracy and detection speed [3, 29, 39].

LB-BCSD methods primarily calculates similarity at the function level and is divided into two stages [20, 45]: 1) Function extraction, where code within binary programs is extracted into functions; 2) Similarity calculation, relying on neural networks to transform functions into semantic vectors, with the distance between these vectors representing the similarity between different functions. Researchers [13,

*Zhe Wang is the corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LCTES '24, June 24, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657572

14, 35, 50] have conducted extensive studies on enhancing LB-BCSD techniques, focusing mainly on the similarity calculation stage and paying less attention to the function extraction stage.

Function inlining [46] is an optimization strategy employed by compilers, whereby the code of a callee function is integrated directly into the caller function. Due to various compilation options (such as O1 and O3) initiating different levels of inlining strategies, significant variations arise in the number of instructions and the control flow structures of functions compiled from the same source [12, 47]. This variation substantially impacts the accuracy of the LB-BCSD model. Research [24] indicates that function inlining can lead to a reduction in accuracy ranging from 30% to 40%. Consequently, the LB-BCSD method requires more refined handling of function inlining to mitigate these impacts.

Function inlining significantly impacts the function extraction phase [3, 32, 58, 62], with previous LB-BCSD methods [8, 13] resorting to inline emulation strategies to tackle issues arising from function inlining. Inline emulation [8, 13], grounded in the principle of normalization, aims to replicate the compiler's inlining behavior to eliminate discrepancies between homologous functions compiled at different optimization levels due to inlining. However, the rules of inline emulation are influenced by the size of the caller function and the callee function, both of which change due to function inlining, rendering inline emulation ineffective in normalizing functions. We discovered that although inline emulation can enhance model accuracy by approximately 10%, significant differences still persist between homologous functions utilizing inline emulation rules due to the complexity of function inlining.

Given that functions are often called multiple times, function inlining integrates the callee function into each call site, leading to duplicate code fragments in the binary program. In contrast to function inlining, code extraction [27] identifies and removes duplicate code from the program, replacing it with corresponding function calls. Consequently, we developed CodeExtract, a system based on code extraction technology. It identifies and extracts duplicate code fragments within binary internal functions through similarity matching, thereby extracting code introduced by function inlining. For code that cannot be extracted, we proactively inline it into the caller, eliminating the differences introduced by function inlining. Compared to inlining emulation, CodeExtract effectively reduces the discrepancies introduced by function inlining and brings about a 20% accuracy improvement to the SOTA LB-BCSD models.

Our main contributions are summarized as follows:

- We explored the impact of function inlining on the accuracy of the LB-BCSD model and demonstrated that function inlining significantly affects the accuracy of the LB-BCSD model. We pointed out that existing inline emulation solutions cannot adequately address the issues posed by function inlining and analyzed the reasons.
- By analyzing the behavior of function inlining, we designed a code extraction-based approach. This approach transforms the problem of identifying inlined functions into a problem of computing duplicate code, identifying and extracting duplicate code fragments within binary internal functions through similarity matching, thereby eliminating discrepancies introduced by function inlining.
- We developed a system named CodeExtract, based on code extraction technology, and conducted experiments on three SOTA LB-BCSD models and 11 real-world applications. The experimental results prove that CodeExtract can bring 20% accuracy improvement to the LB-BCSD models.

2 Mitigating Measures for Function Inlining

Function inlining significantly affects the accuracy of LB-BCSD models, with many LB-BCSD methods [10, 23, 55, 56] indicating their accuracy is impacted by function inlining. To address the discrepancies introduced by function inlining, BinGo [8] first proposed the concept of inlining emulation, which was further expanded by Asm2vec [13]. Inlining emulation aims to mitigate the differences between homologous functions caused by function inlining by selectively inlining callee functions at the binary level. The goal is to maintain consistency in the number of instructions and control flow graphs of functions processed through inlining emulation across different optimization levels. The heuristic rules followed by inlining emulation are as follows:

$$\delta(f_s, f_c) = \frac{length(f_s)}{length(f_c)} \tag{1}$$

Here, f_s and f_c respectively represent the callee function and the caller function, while $length(f_s)$ and $length(f_s)$ indicate the number of instructions in the callee and caller functions, respectively. If the value of δ is less than 0.6, or if the number of instructions in the callee function is fewer than 10, then the inline emulation will proceed to inline the callee function into the caller function.

2.1 Limitations in Inline Emulation

Inlining emulation aims to mimic the compiler's inlining behavior so that, after undergoing inlining emulation, homologous functions compiled with different optimization levels can eliminate the discrepancies caused by function inlining. Through a detailed analysis of the inlining emulation rules, we identified two key points: 1) For caller functions with a larger number of instructions, inlining emulation tends to prefer inlining callee functions into the caller. This means that for the same caller function A, compiled with different compilation options (such as O1 and O3) resulting in a different number of instructions, the version of function A with more instructions may inline more callee functions. This prevents inlining emulation from eliminating the discrepancies introduced by function inlining. 2) The inlining emulation rules primarily target callee functions. Similarly, for the same function A compiled with different options O1 and O3, if A@O1 and A@O3 call different callee functions, the inlining emulation rules will inline different callees accordingly, thus failing to eliminate the discrepancies introduced by function inlining.

3 Our Technique

Existing inline emulation schemes are influenced by the compiler optimization flags when determining whether to inline functions, thus failing to eliminate the differences caused by inlining between homologous functions compiled at different optimization levels. The core idea of our technology is to process homologous functions compiled at various optimization levels in an optimization-level-independent manner. In CodeExtract, we will employ two techniques, code extraction and proproactive inlining, to eliminate the discrepancies introduced by function inlining. For a callee function, after inlining, if duplicate code is present in the program, the code extraction technique is used to extract the duplicate code into corresponding function calls. If no duplicate code exists, the proproactive inlining technique is employed to inline the callee function, which is called only once, into the caller.

In this section, we first introduce the impact of function inlining on binary programs, then explain how code extraction and proproactive inlining techniques can eliminate the differences introduced by function inlining, and finally analyze the feasibility of solving the function inlining issue using code extraction and proproactive inlining techniques.

3.1 When Does Function Inlining Introduce Duplicate Code?

Function inlining involves incorporating the callee function directly into the caller. Given that the prerequisite for code extraction to be effective is that function inlining leads to the presence of duplicate code in the program, this section will analyze the circumstances under which function inlining results in duplicate code, based on the number of times the callee function is called.

3.1.1 Callee Function Called Only Once. Figure 1 illustrates the scenario where a callee function, called only once, undergoes function inlining. In this scenario, Function A consists of two basic blocks, A1 and A2, while Function C comprises four basic blocks, C1-C4, and is called by Function A within the binary program. When inlining Function C, compilers typically do not inline the instructions from basic blocks C1 (the prologue) and C4 (the epilogue) into Function A, as they are primarily responsible for the preservation and restoration of registers. Instead, the compiler inlines basic blocks C2 and C3 into Function A, resulting in a new Function A'.

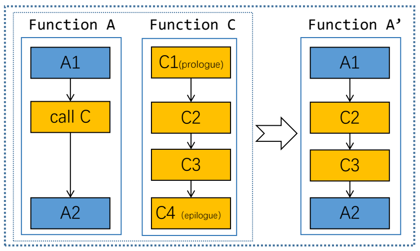


Figure 1. Diagram illustrating the inlining of a function called only once. Function C, exclusively invoked by Function A, is transformed into Function A' following the inlining procedure.

After inlining, since there are no other calls to Function C within the program, compilers usually eliminate the original Function C to reduce the amount of code [46]. At this point, as Function C is inlined only once, its basic blocks C2 and C3 exist solely within Function A', not introducing duplicate code, thus the code extraction technique cannot be applied. For a Function C that is called only once, it may not be inlined under the O1 compilation option, but could be inlined under O3, resulting in differences between Function A@O1 and Function A@O3. To eliminate such discrepancies, we propose a proactive inlining technique, which automatically inlines all functions in the program that are called only once.

3.1.2 Callee Function Called More Than Once. In our previous discussions, we utilized proactive inlining techniques for functions that are called only once. Now, we will analyze the situation when a callee function, called more than once, undergoes function inlining.

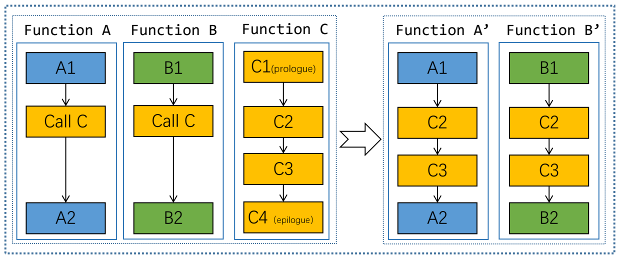


Figure 2. Illustration of the inlining process for functions called more than once, with the inlining of identical sections (basic blocks C2 and C3) of the callee during the process.

Figure 2 depicts a scenario where a callee function is called more than once, leading to the generation of duplicate code. In this example, both Function A and Function B consist of two basic blocks and both call Function C. The compiler inlines Function C into Functions A and B, resulting in the inlined Functions A' and B'. A1, A2, B1, and B2 are the original basic blocks of Functions A and B, while C2 and C3 are basic blocks from Function C. We refer to functions that inline the same basic blocks at different call sites as "inline-stable functions". In this case, both Functions A' and B' contain basic blocks C2 and C3, creating duplicate code. At this point, we can apply code extraction techniques to address these duplicates. If Function C has not been deleted by the compiler, we replace the instructions in basic blocks C2 and C3 with a call to Function C. If Function C has been deleted, the code extraction technique will recreate Function C, place basic blocks C2 and C3 within it, and then replace the basic blocks C2 and C3 in Functions A' and B' with calls to the newly created Function C.

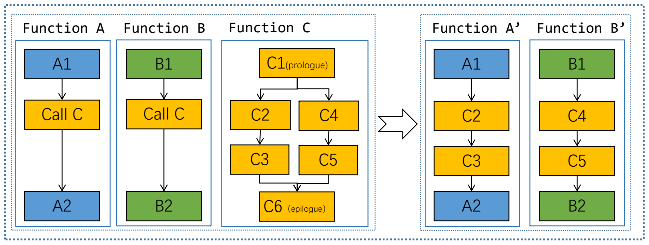


Figure 3. Illustration of the inlining process for functions called more than once, showcasing the inlining of different parts of the callee function. Function A' has inlined basic blocks C1 and C2, while Function B' has inlined basic blocks C4 and C5.

However, even if a callee function is inlined at multiple call sites, it may not result in duplicate code, as illustrated in Figure 3. Unlike the scenario in Figure 2, Function C in this example includes branch instructions. If the branch condition is true, it executes a path containing basic blocks C2 and C3; otherwise, it follows a path with basic blocks C4 and C5. Since the compiler optimizes the inlined code and eliminates dead code, only the executing path's basic blocks, C2 and C3, are retained in Function A', while basic blocks C4 and C5 are not inlined into Function A'. A similar situation applies to Function B', which retains only basic blocks C4 and C5. We call functions that inline different paths at different call sites "inline-sensitive functions". In this case, although Function C is inlined multiple times, since A' and B' retain different basic blocks from Function C, duplicate code is not introduced into the program (unless the body of Function C itself is duplicated). If the body of Function C is deleted, then there is no duplicate code. However, our analysis of real-world programs reveals that this situation is uncommon. In Section 3.2, we will further discuss the practicality and effectiveness of the code extraction scheme from the perspective of real-world programs.

3.2 Feasibility Argument for the Proposed Approaches

In the previous sections, we employed proactive inlining techniques for functions called only once and used code extraction techniques for functions called multiple times. It is important to note that code extraction technology is specifically tailored for handling inlining-stable functions and cannot address inlining-sensitive scenarios. This section delves into a detailed analysis of the inlining phenomenon in actual programs, quantifying the proportion of inlining-sensitive versus inlining-stable functions, and theoretically validating the feasibility of our approach.

Table 1. This table presents the instances of functions being inlined more than once across different programs. The "Num" column represents the number of functions in the program that were inlined more than once. The "Inlining-Stable Num" column displays the count of inlining-stable functions among these, and the "Duplication Ratio (D.R.)" column reflects the proportional relationship between the two.

Programs	Nums	Inlining-Stable Nums	D.R.
blender_r	1767	1671	0.95
cpugcc_r	1727	1538	0.89
cpuxalan_r	2082	2005	0.96
imagick_r	144	132	0.92
leela_r	94	91	0.96
nab_r	36	31	0.86
omnetpp_r	546	526	0.96
perlbench_r	287	246	0.86
povray_r	140	122	0.87
x264_r	137	119	0.87
xz_r	34	33	0.97
Average	**635.82**	**592.18**	**0.92**

We chose the representative SPEC CPU 2017 [6] benchmark suite as our evaluation target. By modifying the function inlining-related code within LLVM [1], we then compiled the C/C++ programs from SPEC CPU 2017 to extract information relevant to function inlining. Our focus was on determining how many functions in SPEC CPU 2017 were inlined more than once, identifying which of these functions were Inlining-Stable, and filtering out programs with fewer than 50 instances of inlining. The results are presented in Table 1.

From Table 1, it is evident that, on average, 635 functions were inlined more than once. Among these, a significant 92% of the functions are inlining-stable, with only 8% being inlining-sensitive. This implies that the code extraction approach can cover 92% of the functions that were inlined multiple times, effectively eliminating the discrepancies introduced by function inlining.

In summary, proactive inlining techniques are suited for functions called only once (as shown in Figure 1), while code extraction techniques are appropriate for functions called multiple times (as depicted in Figure 2). Although code extraction can only handle inline-stable functions among those called multiple times, we have found that in real programs, 92% of functions are inline-stable. Therefore, our proactive inlining and code extraction approach is highly practical and feasible in eliminating the discrepancies introduced by function inlining.

4 Design

Figure 4 illustrates the overall workflow of CodeExtract, comprising two main components: proproactive inlining and code extraction. At the top, we depict the fundamental workflow of the LB-BCSD method. Initially, function extraction is performed on both the target binary program and candidate binary programs. Subsequently, the extracted functions undergo preprocessing to eliminate differences introduced by compilers. Previous studies [8, 13] utilized inline emulation for preprocessing to obtain normalized functions, while this study employs proproactive inlining and code extraction techniques. Following this preprocessing step, the normalized functions serve as inputs to a neural network, which computes the similarity between functions and outputs matched functions.

The input to the CodeExtract system is functions, and the output is normalized functions. Identifying duplicate code relies on similarity calculations at the basic block level, which can be computationally intensive due to the presence of numerous basic blocks in programs. To mitigate computational costs, the Filter module filters basic blocks in functions, identifying basic blocks introduced by function inlining, termed inline basic blocks. Subsequently, similarity calculations are performed on these inline basic blocks to extract highly similar duplicate code, resulting in the output of normalized functions. To provide better elucidation of our design, we first provide the following definitions:

Successors (Succ): In a control flow graph, the successors of a node n are the nodes directly reachable from n.

Predecessors (Pred): In a control flow graph, the predecessors of a node n are the nodes from which n can be directly reached.

Descendants: In a control flow graph, the descendants of a node n are all nodes reachable from n by following a path of edges in the forward direction.

Ancestors: In a control flow graph, the ancestors of a node n are all nodes from which n can be reached by traversing edges in the backward direction.

4.1 Challenge

CodeExtract eliminates discrepancies introduced by function inlining by extracting duplicate code within functions, relying on the calculation of basic block similarity to identify such duplicate code. However, when analyzing the similarity of basic blocks within the same program, we encountered the following challenges:

- Challenge 1: Given that the identification of duplicate code is based on the calculation of similarity at the basic block level, and a program may contain a large number of basic blocks, performing pairwise similarity calculations among all basic blocks can be very time-consuming.
- Challenge 2: When performing function inlining, the compiler optimizes the instructions within the callee function, leading to significant variations in the instructions of the same callee function's basic blocks at different call sites. This results in considerable similarity differences for the same basic block in different contexts.

To address Challenge 1, we designed a Filter module that avoids the need for pairwise similarity calculations across all basic blocks in the program by analyzing all potential "entry point basic blocks" within the control flow graph. To overcome Challenge 2, we normalized the basic blocks before calculating their similarity to reduce the impact of discrepancies caused by inlining. For detailed methods and technical specifics, please refer to Section 4.3.

4.2 Filter Module

Given the vast number of functions in binary programs, each comprising numerous basic blocks, computing similarity for every pair of basic blocks within the program would be exceedingly time-consuming. To circumvent this, the Filter module analyzes the basic blocks within the program, filtering out those not introduced by function inlining. As previously mentioned, functions processed through inlining exhibit a unique characteristic in the control flow graph (CFG): they connect to external instructions solely through a unified entry point, referred to as the "entry basic block." Since only the entry basic block and its descendants could be basic blocks introduced by inlining, limiting similarity calculations to these blocks can significantly reduce computational demands.

The identification of entry basic blocks is based on their distinct characteristic: other basic blocks stemming from an entry basic block can only have the entry basic block or its successor blocks as predecessors. We traverse all basic blocks within a function, utilizing this feature to determine if each block is a potential entry basic block. Initially, we consider each basic block as a potential entry basic block and exhaustively enumerate all control flow subgraphs starting from it within the CFG. Subsequently, we assess whether these basic blocks align with the characteristics of an entry basic block, a process elaborately described in Algorithm 1. This algorithm accepts a function extracted from a binary program as input

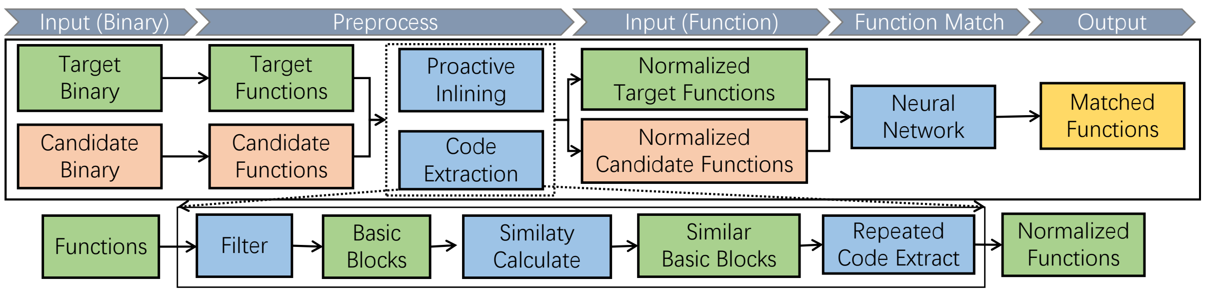


Figure 4. Workflow of CodeExtract: At the top is the workflow of the LB-BCSD method. Our technique involves proactive inline and code extrication.

Algorithm 1: Identifying Entry Basic Blocks

```
   Input: Functions, denoting all functions within the
          program.
   Output: entryBB, denoting all entry basic blocks.
1  entryBB ← {}
2  for each func ∈ functions do
3  |   for each bb ∈ func do
4  |   |   if isEntryBB(bb, k=3) then
5  |   |   |   entryBB.append(bb)
6  return entryBB
7
8  function isEntryBB(bb,k)
9  |   subgraphs ← Enumerate control flow subgraphs
   |     starting with BB, with subgraph size k
10 |   for each subgraph ∈ subgraphs do
11 |   |   Set checkFlag := True
12 |   |   for each node ∈ subgraph do
13 |   |   |   if bb ∉ node.ancestors then
14 |   |   |   |   Set checkFlag := False
15 |   |   |   |   Break
16 |   |   |   if node.predecessors ∉ subgraph then
17 |   |   |   |   Set checkFlag := False
18 |   |   |   |   Break
19 |   |   if checkFlag then
20 |   |   |   return True
21 |   return False
```

and outputs all identified potential entry basic blocks. While traversing each basic block, we employ the isEntryBB(BB, k) function to determine its conformity to the characteristics of an entry basic block and return all possible entry basic blocks (lines 1-6). In this function, we first exhaustively enumerate all control flow subgraphs starting from BB, limiting the size of the subgraph to k basic blocks to ensure that the control flow subgraph of the identified entry basic block contains at least k basic blocks. If the number of basic blocks introduced into the caller by an inlined callee function is less than k, then these less numerous callee functions' entry points will not be recognized as entry basic blocks.

For each control flow subgraph, we further traverse each node, checking if its predecessor is the entry basic block or its descendants. As long as there is a control flow subgraph that makes a basic block meet the conditions of an entry basic block, we consider it as such (lines 10-21). In practice, we set k=3, and in Section 5.3, we discuss the impact of the number of basic blocks in a function on the accuracy of the LB-BCSD model. Functions with fewer than 3 inline basic blocks have a negligible impact on the accuracy of the LB-BCSD method, thus obviating the need to extract these less frequent duplicate codes.

Once the entry basic blocks are identified, we only need to mark their direct successors as the basic blocks to be matched, rather than all descendants. This is because, in the subsequent duplicate code extraction algorithm, given a pair of similar basic blocks, we continue to match the similarity of other basic blocks near this pair as starting points. This approach significantly reduces the number of basic blocks to be matched, thereby effectively decreasing the computational overhead of basic block similarity calculations.

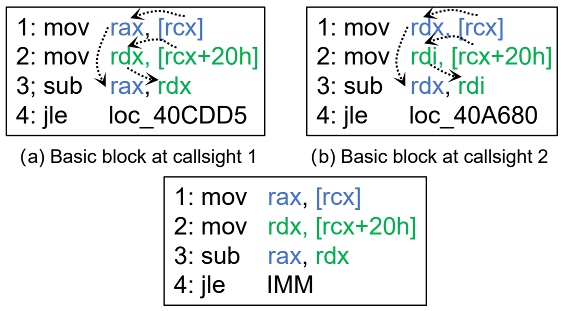
4.3 Similarity Calculate Module

After acquiring the basic blocks to be matched, the subsequent task involves calculating the similarity between these blocks. In the domain of LB-BCSD, computing similarity at the basic block level poses a significant challenge. This is primarily because, compared to functions, basic blocks typically contain fewer instructions, making it difficult for LB-BCSD methods to generate accurate semantic vectors for basic blocks. To address this challenge, SOTA LB-BCSD [14, 41, 57, 63] approaches leverage the contextual information of basic blocks to assess their similarity, a strategy that has shown effective results in matching basic blocks across different programs.

However, the main objective of this paper is to identify duplicate code within the same binary program. In this scenario, function inlining results in the callee function being inlined into different call sites, causing originally similar basic blocks to be placed in entirely different contexts. Thus, existing LB-BCSD methods based on contextual information at the basic block granularity are not suitable for addressing the basic block matching problem within the same binary program.

We observe that within the same binary program, when a callee function is inlined into different call sites, its control flow structure and instructions largely remain unchanged. This observation leads us to treat the instructions in the basic blocks as strings and evaluate the similarity of basic blocks by calculating the edit distance between these instruction strings. Edit distance [43, 53] (also known as Levenshtein distance) measures the minimum number of single-character editing operations required to transform one string into another (including insertion, deletion, and substitution). A smaller edit distance implies closer functional proximity of the basic blocks.

Empirically, we consider basic blocks with a similarity exceeding 0.95 to be similar. However, in practice, optimizations by the compiler during function inlining may change the register names in the callee functions, increasing the edit distance between two similar basic blocks. This paper will further explore the root causes of this issue and propose corresponding strategies to improve the accuracy of duplicate code identification within the same binary program.


```
1: mov  rax, [rcx]
2: mov  rdx, [rcx+20h]
3; sub  rax, rdx
4: jle  loc_40CDD5
```
(a) Basic block at callsight 1

```
1: mov  rdx, [rcx]
2: mov  rdi, [rcx+20h]
3: sub  rdx, rdi
4: jle  loc_40A680
```
(b) Basic block at callsight 2

```
1: mov  rax, [rcx]
2: mov  rdx, [rcx+20h]
3: sub  rax, rdx
4: jle  IMM
```
(c) Normalized basic block

Figure 5. Due to compiler optimizations, the same basic block of a callee function utilizes different registers at various call sites (a) and (b). Through data flow analysis, these basic blocks from different call sites have been normalized, as illustrated in figure (c).

4.3.1 Differences Introduced by Variations in Register Names.

In Figure 5, the same basic block of the callee function is inlined at different call sites 1 and 2. Due to the compiler's optimization that merges the callee and caller, different registers are utilized at different call sites [46]. This variance due to compiler optimization can be mitigated by renaming the registers within the basic block. This process is achieved through def-use analysis of the registers, where registers exhibiting identical def-use behavior are assigned the same name. In Figure 5(a), we observe that call site 1 primarily performs def operations on *rax* and *rdx*, with *rax*'s value originating from memory pointed to by *rcx* and serving as the first operand of the *sub* instruction; *rdx*'s value comes from memory at $rcx + 20$ and serves as the *sub* instruction's second operand. In Figure 5(b), *rdx* exhibits the same def-use behavior as *rax* in Figure 5(a), while *rdi* behaves similarly to *rdx* in Figure 5(a). Hence, we rename the registers used in Figure 5(b), with the results shown in Figure 5(c). Additionally, to eliminate differences introduced by varying jump instruction targets due to different calling contexts, jump instruction targets are uniformly replaced with the *IMM* label.

4.4 Repeated Code Extract Module

After completing the similarity calculations for basic blocks, we identify similar basic blocks. For instance, if basic block A is similar to basic blocks B and C, this suggests that ABC might be the result of the same function being inlined at different call sites. To identify all inlined basic blocks, CodeExtract does not immediately extract code from these similar basic blocks. Instead, based on control flow, it continues to analyze the similarity of neighboring basic blocks to the similar ones until no new similar basic blocks are identified, and only then does it perform the extraction of duplicate code. For example, in Figure 2, once we confirm that the basic block C2 in function A' and function B' are similar, we mark the basic block C2 as duplicate code. Next, we assess the predecessors and successors of these duplicate codes—whether the predecessor and successor nodes of the basic block C2 are similar. This search process continues until no more similar basic blocks are found. This entire process is described by Algorithm 2.

Algorithm 2 takes as input a function f and a set of similar basic blocks fbb within function f, and outputs the modified function f'. This algorithm iterates through each basic block tbb in fbb, searching for all instances of duplicate code starting from tbb, and then removes these codes from function f (lines 1-8), inserting a call instruction at the appropriate location to invoke the removed duplicate code. The *FindRepeatedCode* function takes two basic blocks tbb and sbb as inputs, aiming to recursively search for similar basic blocks within the predecessors and successors of tbb and sbb, adding them to the *RepeatedCode* collection of duplicate codes (lines 12-20). If the number of basic blocks in the *RepeatedCode* collection is less than k, it returns an empty set; otherwise, it returns the *RepeatedCode* collection. Here, sim refers to the edit distance mentioned earlier, for which we empirically set $\gamma = 0.95$. In practice, we set $k = 3$, and in Table 3, we demonstrate that functions with fewer than 3 basic blocks have a negligible impact on the accuracy of the LB-BCSD model. Therefore, during code extraction, we focus only on those inlined functions with a number of basic blocks greater than or equal to 3.

Algorithm 2: Identifying Duplicate Code

```
Input: $f$: Target Function, $fbbs$: similar basic blocks
       belongs to $f$
Output: Extracted Function f'
$repeatedCode := set()$
for each $tbb \in fbbs$ do
    $sbbs$ := basic blocks similar to $tbb$
    for each $sbb \in sbbs$ do
        $code$ := findRepeatedCode($tbb, sbb$)
        if $code \neq []$ then
            $repeatedCode.add(code)$
$f' := f - repeatedCode$
return (f')

Function findRepeatedCode($tbb, sbb$):
    $repeatedCode = [tbb]$
    $bk\_tbb := tbb$
    while $sim(tbb.pred, sbb.pred) > \gamma$ do
        $repeatedCode.add(tbb.pred)$
        $tbb := tbb.pred$
        $sbb := sbb.pred$
    $tbb := bk\_tbb$
    while $sim(tbb.succ, sbb.succ) > \gamma$ do
        $repeatedCode.add(tbb.succ)$
        $tbb := tbb.succ$
        $sbb := sbb.succ$
    if $repeatedCode.size < k$ then
        return []
    return $repeatedCode$
```

5 Evaluation

CodeExtract is implemented on the angr [49] platform. To comprehensively evaluate our technique, we have set the following research questions (RQs) for in-depth exploration:
RQ1: How much does function inlining affect the accuracy of the LB-BCSD model?
RQ2: What is the impact of the number of basic blocks in a function on the LB-BCSD model?
RQ3: Can CodeExtract significantly improve the accuracy of the LB-BCSD model when facing function inlining?
RQ4: why CodeExtract can enhance the accuracy of the LB-BCSD model?

5.1 Experiment Setup

We conduct the experiments on a server with a Intel Xeon Gold 6132 CPU at 2.60GHz, 256 GB memory, 8 Tesla V100 GPUs, and Ubuntu 18.04.

5.1.1 Baseline Models. We selected three SOTA LB-BCSD models as baselines: a) SAFE [35] employs an RNN structure with a self-attention mechanism to generate semantic embeddings for functions; b) Asm2vec [13] utilizes the PV-DM model combined with the program's control flow information to generate semantic embeddings for functions; c) JTrans [50] encodes control flow information into a Transformer architecture to produce semantic embeddings for functions. These three models are open-source, and we utilized the implementations provided by the authors of the papers for the binary similarity matching task.

5.1.2 Test Datasets. The SPEC CPU 2017 suite, which gathers programs covering a broad spectrum of computational and application scenarios, has emerged as the benchmark of choice for numerous research studies [42]. Leveraging this, we selected C/C++ programs from the SPEC CPU 2017 [6] test suite to create our dataset, excluding those with fewer than 50 instances of function inlining. As a result, our dataset encompasses 11 real-world projects, including blender_r, cpugcc_r, cpuxalan_r, imagick_r, leela_r, nab_r, omnetpp_r, perlbench_r, povray_r, x264_r, and xz_r. In alignment with previous research [13, 24, 50], we utilized programs compiled with the O1 and O3 optimization options of llvm-10.0 [1] for our test set.

5.1.3 Metrics. In this experiment, we employed the top-10 accuracy as our evaluation metric. This metric focuses on whether the correct answer (i.e., the function compiled from the same source code as the queried binary function) appears within the top 10 ranked options among all predictions when iteratively querying a set of binary functions from the candidate function pool. Furthermore, we also conducted an analysis on the false positive and false negative rates of CodeExtract.

5.2 RQ1: Impact of Function Inlining on Model Accuracy

In this section, we evaluate the impact of function inlining on model accuracy. For each baseline model, we input binary programs with function inlining enabled and disabled, respectively. By comparing the accuracy changes of the models with function inlining turned on and off, we represent the influence of function inlining on model accuracy. For every binary program in our dataset, we randomly sample 500 functions from the O0 binary and query them one by one in a pool composed of the corresponding 500 functions from the O3 binary. In other words, for each query, there is only one similar function in the function pool. This setup is consistent with the literature [22, 33, 52, 54].

The results are shown in Table 2. We can see that LB-BCSD models exhibit high top-10 accuracy for programs without function inlining, with JTrans reaching an accuracy of up to 84%. This demonstrates that existing LB-BCSD

Table 2. The impact of function inlining on model accuracy. NI represents the top-10 accuracy of the model with inlining disabled, WI represents the top-10 accuracy of the model with inlining enabled, and Infl. denotes the influence of function inlining on model accuracy.

Programs	Asm2vec			SAFE			JTrans		
	NI	**WI**	**Infl.**	**NI**	**WI**	**Infl.**	**NI**	**WI**	**Infl.**
blender_r	0.86	0.56	0.3	0.77	0.55	0.22	0.88	0.61	0.27
cpugcc_r	0.78	0.51	0.27	0.71	0.46	0.25	0.83	0.60	0.23
cpuxalan_r	0.74	0.43	0.31	0.71	0.42	0.29	0.79	0.50	0.29
imagick_r	0.88	0.55	0.33	0.85	0.54	0.31	0.86	0.51	0.35
leela_r	0.74	0.34	0.40	0.76	0.43	0.33	0.81	0.45	0.36
nab_r	0.83	0.52	0.31	0.79	0.53	0.26	0.85	0.50	0.35
omnetpp_r	0.72	0.31	0.41	0.75	0.43	0.32	0.77	0.38	0.39
perlbench_r	0.89	0.65	0.24	0.84	0.57	0.27	0.86	0.52	0.34
povray_r	0.86	0.57	0.29	0.72	0.47	0.25	0.88	0.63	0.25
x264_r	0.83	0.56	0.27	0.76	0.54	0.22	0.85	0.56	0.29
xz_r	0.91	0.54	0.37	0.73	0.44	0.29	0.92	0.59	0.33
Average	0.82	0.50	0.32	0.76	0.49	0.27	0.84	0.53	0.31

models can correctly identify the relationships between instructions and accurately extract the semantic information of functions. However, the accuracy of existing LB-BCSD models decreases by about 31% for programs with function inlining, with JTrans having the highest accuracy at 53%. This result is consistent with the literature [24], indicating that function inlining significantly impacts the precision of LB-BCSD models. Compared to JTrans and Asm2vec, the impact of function inlining on SAFE is minimal, only 27%. This is mainly because function inlining significantly affects the control flow structure of programs, and Asm2vec and JTrans encode the program's control flow information into the function semantic vector, whereas SAFE does not, thus making its accuracy less impacted by function inlining. In summary, function inlining significantly affects the accuracy of LB-BCSD models, causing an average accuracy drop of 30%.

5.3 RQ2: Impact of Callee Function Size on LB-BCSD Model Accuracy

In this experiment, we investigated the impact of the callee function's basic block count on the accuracy of the LB-BCSD model. To this end, we modified the LLVM source code while keeping the original inlining rules unchanged. That is, when the compiler decides to inline a function, we check the number of basic blocks in that function. If the count is less than or equal to X, then it is inlined; otherwise, it is not.

The experimental results, as shown in Table 3, reveal that when function inlining is disabled, the accuracy of the LB-BCSD model is 84%. However, when the compiler only inlines those functions with a basic block count of two or less, the accuracy of the LB-BCSD model decreases by 3%. This decline is primarily because smaller functions lack complex control flow structures and have fewer instructions, thus having a minimal impact on the accuracy of the LB-BCSD model. Conversely, when the compiler only inlines functions with a basic block count of three or less, the accuracy of the LB-BCSD model drops by 15%. From this, we can infer that the accuracy of the LB-BCSD is significantly affected only when the inlined functions have a basic block count of three or more.

Table 3. Analyzing the impact of the number of basic blocks in functions with function inlining on the Top-10 accuracy of the JTrans model. "NI" stands for function inlining disabled, "WI" represents function inlining enabled, and "BBX" indicates that functions will not be inlined if the number of basic blocks exceeds X.

Programs	**NI**	**BB1**	**BB2**	**BB3**	**BB5**	**BB10**	**BB15**	**WI**
blender_r	0.88	0.87	0.85	0.75	0.68	0.65	0.63	0.61
cpugcc_r	0.83	0.82	0.79	0.73	0.69	0.64	0.62	0.60
cpuxalan_r	0.79	0.78	0.75	0.65	0.57	0.54	0.52	0.50
imagick_r	0.86	0.85	0.83	0.71	0.68	0.61	0.57	0.51
leela_r	0.81	0.79	0.77	0.55	0.49	0.46	0.44	0.45
nab_r	0.85	0.84	0.83	0.68	0.61	0.57	0.55	0.50
omnetpp_r	0.77	0.75	0.73	0.59	0.45	0.41	0.38	0.38
perlbench_r	0.85	0.84	0.81	0.71	0.69	0.67	0.66	0.52
povray_r	0.87	0.86	0.85	0.66	0.63	0.58	0.57	0.57
x264_r	0.85	0.84	0.82	0.73	0.71	0.65	0.59	0.56
xz_r	0.92	0.90	0.88	0.83	0.81	0.77	0.75	0.59
Average	**0.84**	**0.83**	**0.81**	**0.69**	**0.64**	**0.60**	**0.57**	**0.53**

5.4 RQ3: Accuracy Enhancements Brought to LB-BCSD Models by CodeExtract

In this section, we evaluated the accuracy improvements brought by inline emulation and CodeExtract to the models. Since inline emulation algorithms are only implemented in Asm2vec, to fairly assess the impact of inline emulation on Asm2vec, SAFE, and JTrans, we modified the function inlining-related code in LLVM. When compiling the SPEC benchmarks, functions were inlined according to the rules of inline emulation discussed earlier. The other settings in this experiment are the same as in Section 5.2.

The experimental results, as shown in Table 4, indicate that inline emulation can bring about a 9%-12% accuracy improvement for the three models. In comparison, code extraction can lead to an 18%-22% accuracy improvement for the three models. Inline emulation can, to some extent, make the homologous functions compiled with O1 and O3 more similar. We discussed the drawbacks of inline emulation in the section 2.1, so it cannot resolve the issues introduced by function inlining. On the other hand, our CodeExtract can extract the inlined basic blocks introduced by function inlining, making the function's functionality relatively pure, thus better eliminating the discrepancies introduced by function inlining. In the following experiments, we will systematically evaluate the advantages and disadvantages of inline emulation and CodeExtract in terms of the number of instructions, false positive rates, and false negative rates.

Table 4. This table shows the impact of two techniques, inline emulation (IE) and CodeExtract (CE), on model accuracy. Ori. represents the original top-10 accuracy of the model. IE/CE represent the top-10 accuracy of the model after applying Inline Emulation and CodeExtract techniques, respectively. Correspondingly, IE/CE Impr. shows the specific improvement of these two techniques on model accuracy.

Programs	**Asm2vec**					**SAFE**					**JTrans**				
	Ori.	**IE**	**IE Impr.**	**CE**	**CE Impr.**	**Ori.**	**IE**	**IE Impr.**	**CE**	**CE Impr.**	**Ori.**	**IE**	**IE Impr.**	**CE**	**CE Impr.**
blender_r	0.56	0.65	0.09	0.79	0.19	0.55	0.65	0.1	0.76	0.17	0.61	0.74	0.13	0.83	0.22
cpugcc_r	0.51	0.59	0.08	0.77	0.21	0.46	0.54	0.08	0.71	0.22	0.60	0.71	0.11	0.83	0.23
cpuxalan_r	0.43	0.55	0.12	0.61	0.18	0.42	0.53	0.11	0.61	0.19	0.50	0.65	0.15	0.69	0.19
imagick_r	0.55	0.68	0.13	0.79	0.20	0.54	0.65	0.11	0.75	0.21	0.51	0.65	0.14	0.74	0.23
leela_r	0.34	0.42	0.08	0.65	0.27	0.43	0.49	0.06	0.68	0.23	0.45	0.53	0.08	0.74	0.29
nab_r	0.52	0.63	0.11	0.70	0.18	0.53	0.62	0.09	0.69	0.16	0.50	0.61	0.11	0.75	0.25
omnetpp_r	0.31	0.38	0.07	0.52	0.21	0.43	0.51	0.08	0.60	0.17	0.38	0.47	0.09	0.56	0.18
perlbench_r	0.65	0.78	0.13	0.92	0.27	0.57	0.67	0.10	0.85	0.17	0.52	0.64	0.12	0.80	0.28
povray_r	0.57	0.66	0.09	0.71	0.14	0.47	0.55	0.08	0.60	0.13	0.63	0.74	0.11	0.87	0.24
x264_r	0.56	0.68	0.12	0.76	0.20	0.54	0.64	0.10	0.71	0.17	0.56	0.69	0.13	0.74	0.18
xz_r	0.54	0.65	0.11	0.76	0.19	0.44	0.53	0.09	0.62	0.18	0.59	0.73	0.14	0.78	0.19
Average	0.50	0.61	0.10	0.73	0.20	0.49	0.58	0.09	0.69	0.18	0.53	0.65	0.12	0.76	0.23

5.5 RQ4: False Positive and False Negative Analysis

In Section 5.3, we discovered that the presence of three or more basic blocks in inlined functions significantly influences the accuracy of the LB-BCSD model. Consequently, this experiment aimed to analyze the rates of false positives and false negatives for both inlining emulation and CodeExtract across all functions and specifically those with a basic block count of three or more in the respective programs. To ensure the precision of our assessment, we benchmarked against the compilation outcomes obtained using LLVM with the O3 optimization level.

Table 5. This figure presents an analysis of the false positive rate (FPR) and false negative rate (FNR) for both inline emulation and CodeExtract methods on program functions. In the table, "Functions" refers to all functions within a program, while "Functions_BB3" specifically denotes functions with a number of basic blocks greater than or equal to 3.

Programs	Inline Emulation				CodeExtract			
	Functions		Functions-BB3		Functions		Functions-BB3	
	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR
blender_r	0.77	0	0.7	0.27	0.79	0.05	0.42	0.03
cpugcc_r	0.78	0.13	0.8	0.15	0.83	0.17	0.55	0.05
cpuxalan_r	0.84	0.1	0.67	0.18	0.86	0.15	0.51	0.09
imagick_r	0.73	0.11	0.95	0.08	0.96	0.16	0.65	0.02
leela_r	0.82	0.09	0.68	0.18	0.76	0.19	0.49	0.19
nab_r	0.81	0.05	0.69	0.15	0.91	0.15	0.44	0.02
omnetpp_r	0.95	0.05	0.74	0.27	0.78	0.17	0.59	0.12
perlbench_r	0.78	0.18	0.8	0.1	0.94	0.24	0.68	0.02
povray_r	0.78	0.09	0.59	0.15	0.92	0.17	0.62	0.03
x264_r	0.78	0.14	0.71	0.14	0.87	0.16	0.59	0.04
xz_r	0.79	0.07	0.63	0.33	0.76	0.19	0.78	0.12
Average	**0.8**	**0.09**	**0.72**	**0.18**	**0.85**	**0.16**	**0.57**	**0.07**

The experimental results, presented in Table 5, demonstrate that CodeExtract exhibits a higher rate of both false positives and false negatives for all functions within the respective programs compared to inlining emulation. The increased false negative rate is attributed to CodeExtract's method of only extracting functions with a basic block count of three or more, ignoring those with fewer than three blocks. The rise in false positives is due to CodeExtract's proactive inlining strategy, which inlines all functions called only once, leading to a higher incidence of false positives. However, CodeExtract shows lower rates of false positives and false negatives for functions with a basic block count of three or more, indicating its superior handling of such functions. Given that the accuracy of the LB-BCSD model is significantly affected only when functions with three or more basic blocks are inlined, employing CodeExtract proves to be more efficacious in enhancing the LB-BCSD model's accuracy compared to relying solely on inlining emulation.

6 Conclusion

Function inlining significantly affects the accuracy of the LB-BCSD model, and previous methods based on inline emulation have not been able to fully address the issues introduced by function inlining. Therefore, we propose a system called CodeExtract, based on code extraction and proactive inlining techniques. Compared to inline emulation, CodeExtract can better eliminate the discrepancies introduced by function inlining. Experiments have shown that, in addressing function inlining issues, CodeExtract can improve the accuracy of the LB-BCSD model by 20%.

Acknowledgments

This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 61902374, 62272442, U1736208, and the Innovation Funding of ICT, CAS under Grant No.E161040.

References

[1] 2020. LLVM. *clang-10. Retrieved Feb 16, 2023 from https://releases.llvm.org/10.0.* (2020).

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202

[3] Saed Alrabaee, Mourad Debbabi, and Lingyu Wang. 2022. A survey of binary code fingerprinting approaches: taxonomy, methodologies, and features. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–41.

[4] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[5] 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.

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[7] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2013), 307–317.

[8] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.

[9] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *Acm Sigplan Notices* 51, 6 (2016), 266–280.

[10] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices* 53, 2 (2018), 392–404.

[11] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[12] Jack W Davidson and Anne M Holler. 1988. A study of a C function inliner. *Software: Practice and Experience* 18, 8 (1988), 775–790.

[13] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[14] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*.

[15] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Sstic* 5, 1 (2005), 3.

[16] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*. 303–317.

[17] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. 2014. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 78–87.

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

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

[20] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–38.

[21] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 57–67.

[22] 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 *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 104–114. https://doi.org/10.1109/ICSME.2018.00019

[23] He Huang, Amr M Youssef, and Mourad Debbabi. 2017. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 155–166.

[24] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 2023. 1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.

[25] Lichen Jia, Yang Yang, Jiansong Li, Hao Ding, Jiajun Li, Ting Yuan, Lei Liu, and Zihan Jiang. 2023. MTMG: A Framework for Generating Adversarial Examples Targeting Multiple Learning-Based Malware Detection Systems. In *Pacific Rim International Conference on Artificial Intelligence*. Springer, 249–261.

[26] Lichen Jia, Yang Yang, Bowen Tang, and Zihan Jiang. 2023. ERMDS: A obfuscation dataset for evaluating robustness of learning-based malware detection system. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 3, 1 (2023), 100106.

[27] Raghavan Komondoor and Susan Horwitz. 2003. *Eliminating duplication in source code via procedure extraction.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[28] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309. https://doi.org/10.1109/WCRE.2001.957835

[29] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251.

[30] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

[31] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. https://doi.org/10.1109/TSE.2017.2655046

[32] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS*.

[33] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.

[34] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. arXiv:2103.11626 [cs.SE]

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

[36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. {BinSim}: Trace-based Semantic Binary Diffing via System Call Sliced

Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. 253–270.

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

[38] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. 2017. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 341–355.

[39] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs?. In *International Conference on Machine Learning*. PMLR, 8476–8486.

[40] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 406–415. https://doi.org/10.1145/2664243.2664269

[41] Kimberly Redmond, Lannan Luo, and Qiang Zeng. 2018. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652* (2018).

[42] 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.

[43] Eric Sven Ristad and Peter N Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (1998), 522–532.

[44] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[45] Samuel Henrique Silva and Peyman Najafirad. 2020. Opportunities and challenges in deep learning adversarial robustness: A survey. *arXiv preprint arXiv:2007.00753* (2020).

[46] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989.

[47] Jon Tschudi, Marion O'Farrell, and Kari Anne Hestnes Bakke. 2018. Inline spectroscopy: From concept to function. *Applied Spectroscopy* 72, 9 (2018), 1298–1309.

[48] Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *SIGAPP Appl. Comput. Rev.* 19, 4 (jan 2020), 28–39. https://doi.org/10.1145/3381307.3381310

[49] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

[50] 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 detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13.

[51] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.

[52] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.

[53] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment* 1, 1 (2008), 933–944.

[54] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. (2023).

[55] 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 2017 ACM SIGSAC conference on computer and communications security*. 363–376.

[56] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884.

[57] 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* (2021).

[58] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep-Learning Based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Transactions on Software Engineering and Methodology* (2023).

[59] 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 *2020 IEEE Symposium on Security and Privacy (SP)*. 1121–1138. https://doi.org/10.1109/SP40000.2020.00035

[60] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

[61] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2023. Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 55–67.

[62] Xiaoya Zhu, Junfeng Wang, Zhiyang Fang, Xiaokang Yin, and Shengli Liu. 2022. BBDetector: A Precise and Scalable Third-Party Library Detection in Binary Executables with Fine-Grained Function-Level Features. *Applied Sciences* 13, 1 (2022), 413.

[63] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).

Received 2024-02-29; accepted 2024-04-01