Xu, Xi; Zheng, Qinghua; Yan, Zheng; Fan, Ming; Jia, Ang; Liu, Ting

Interpretation-enabled Software Reuse Detection Based on a Multi-Level Birthmark Model

Published in:
Proceedings of 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021

DOI:
10.1109/ICSE43902.2021.00084

Published: 07/05/2021

Document Version
Peer reviewed version

Please cite the original version:
Interpretation-enabled Software Reuse Detection Based on a Multi-Level Birthmark Model

Xi Xu *, Qinghua Zheng *, Zheng Yan ††, Ming Fan ††, Ang Jia ††, and Ting Liu ††
*Key Laboratory of Intelligent Networks and Network Security, Ministry of Education, China
†School of Computer Science and Technology, Xi’an Jiaotong University, China
‡School of Cyber Science and Engineering, Xi’an Jiaotong University, China
§State Key Lab on Integrated Services Networks, School of Cyber Engineering, Xidian University, China
†Department of Communications and Networking, Aalto University, Finland

Corresponding author: Zheng Yan.

Abstract—Software reuse, especially partial reuse, poses legal and security threats to software development. Since its source codes are usually unavailable, software reuse is hard to be detected with interpretation. On the other hand, current approaches suffer from poor detection accuracy and efficiency, far from satisfying practical demands. To tackle these problems, in this paper, we propose ISRD, an interpretation-enabled software reuse detection approach based on a multi-level birthmark model that contains function level, basic block level, and instruction level. To overcome obfuscation caused by cross-compilation, we represent function semantics with Minimum Branch Path (MBP) and perform normalization to extract core semantics of instructions. For efficiently detecting reused functions, a process for “intelligent search based on anchor recognition” is designed to speed up reuse detection. It uses strict instruction match and identical library call invocation check to find anchor functions (in short anchors) and then traverses neighbors of the anchors to explore potentially matched function pairs. Extensive experiments based on two real-world binary datasets reveal that ISRD is interpretable, effective, and efficient, which achieves 97.2% precision and 94.8% recall. Moreover, it is resilient to cross-compilation, outperforming state-of-the-art approaches.

Index Terms—Binary Similarity Analysis, Software Reuse Detection, Multi-Level Software Birthmark, Interpretation

I. INTRODUCTION

Along with the growing popularity of open-source software, software reuse becomes a common phenomenon. However, extensive reuse of existing codes leads to numerous license violation issues [1], [2]. For example, Cisco and VMware were exposed to significant legal issues because they did not adhere to the licensing terms of Linux kernel [3], [4]. What is more, security issues could be raised due to careless software reuse [5].

The goal of software reuse detection is to determine whether a candidate program contains similar codes already used in a target program. There are many approaches [6], [7] proposed in the literature. According to the analysis objects, these approaches can be divided into two groups: source code reuse detection and binary code reuse detection. The first calculates code similarity by abstracting source codes into a set of characteristics, such as string [8], [9], token [10], [11], [12], Abstract Syntax Tree (AST) [14], [15], [16], [17] and Program Dependency Graph (PDG) [18], [19], [20]. Since the source code of a candidate program is typically unavailable in reality, binary code reuse detection is used widely for software plagiarism detection [21], [22], [23], malware detection [24], [25], [26], patch analysis [27], [28], [29], and so on.

However, the existing binary code reuse detection approaches suffer from three limitations, as described below.

Poor interpretability: The detection results of the existing approaches [21], [24], [27] lack the ability to provide detailed evidence to support reuse detection results because they usually only report their results in form of similarity scores ranging from 0 to 1. The ability to comprehensively interpreting the detection results is extremely important since it can provide the details or reasons to make the detection results acceptable or easy to be understood.

Poor accuracy: Most approaches [30], [31], [32], [33], [34] that rely on structural and syntax information fail to deal with the differences caused by variations in compilation. This is because different compilations of a source program naturally produce different structures and syntax in its binary codes, forming obfuscation. For example, the approaches proposed in [35], [36] that operate at the boundaries of a basic block might fail to deal with basic block splitting or merging.

Poor efficiency: Several works [34], [35] use a brute force method to identify reused functions, which is prohibitively expensive since it measures the similarities of all function pairs between a target program and a candidate program. Such approaches lead to poor efficiency if the numbers of functions in both programs are big. Therefore, these approaches are neither effective in case that a reused part only makes up a small percentage of the candidate program, nor efficient if an excessive number of function pairs need to be compared.

To overcome the above limitations, we propose ISRD, a novel Interpretation-enabled Software Reuse Detection approach based on a multi-level birthmark model, which holds the following salient advantages:

Interpretable detection results. ISRD is capable of capturing program semantics from coarse granularity to fine granularity and uniquely identifying a program with a multi-level birth-
mark model that contains function level, basic block level, and instruction level. Specifically, at the function level, a Function Call Graph (FCG) is constructed to profile program behavior. The FCG of a program is a directed graph, which consists of a set of nodes representing functions and a set of edges representing caller and callee relationships among functions [37]. Then, basic block chains and normalized instructions are extracted to demonstrate the semantics of the function at the basic block level and the instruction level, respectively. The similar parts between the birthmark of a target program and that of a candidate program in the above three levels can reconstruct a reuse scene to interpret and justify detection results. Obviously, this kind of demonstration can assist easy understanding and trust on a detection result, unlike a simple value reuse indicator.

High accuracy. To achieve high accuracy of reuse detection, we perform normalization at different levels of our birthmark model. Specifically, at the basic block level, we first transform each function into a set of Minimum Branch Paths (MBPs), which are length variant partial execution paths starting from an initial node or a branch node and ending at a terminal node or an adjacent branch node. Then, to mitigate huge differences in instructions caused by cross-compilation, we only consider the key instructions to represent their core semantics. Furthermore, we lift low-level assembly instructions up to high-level operations and remove operands from them to address syntax differences of instructions to complete the process of instruction normalization.

High efficiency. To speed up the similarity calculation among thousands of function pairs, we propose a process for “intent search based on anchor recognition” to first recognize anchor functions (in short anchors) and then perform intent search originated from the anchors to discover all potentially matched function pairs. In this way, we significantly reduce the number of comparisons before similarity calculation and meanwhile ensure comparison quality. Concretely, the process leverages on strict instruction match and identical library call invocation check to search for matched function pairs as anchors by considering both developer-defined functions and library functions. Through intent search originated from the anchors, it further explores neighbors of the anchors to find new function pairs with high similarity scores.

Moreover, since there is currently no dataset that can be directly used for partial reuse detection tests, we construct a dataset that contains 24 real-world software projects and manually label a total of 74 partial reuses. The dataset has been published on website [38]. By evaluating ISRD based on our constructed dataset and a widely used dataset, we demonstrate that ISRD exhibits impressive software reuse detection performance. In summary, the major contributions of this paper include:

(i) We propose a novel fine-granular multi-level birthmark model to uniquely represent program semantics and enable interpretability of software reuse detection result.
(ii) We perform normalization at both the basic block level and the instruction level to resist semantics-preserving obfuscation for accurate software reuse detection.
(iii) We design a process to recognize anchors and conduct intent search originated from the anchors, which can greatly reduce the number of comparisons, thus significantly accelerate detection speed.
(iv) We implement ISRD and evaluate its performance with extensive experiments. The results reveal that ISRD is interpretable, can effectively and efficiently detect partial reuse. It is also resilient to cross-compilation, outperforming the state-of-the-art approaches.

II. RELATED WORK

According to analysis techniques, existing binary code reuse detection approaches can be divided into two main categories: static analysis and dynamic analysis.

Static Analysis. Static analysis is applied on binaries without running programs. BinDiff [34] was proposed to use the structural similarity of CFG to compare binary codes. Luo et al. [21] proposed CoP, a binary-oriented, obfuscation-resilient method for code reuse detection, which combines rigorous program semantics with the longest common subsequence based fuzzy matching. David et al. [36] proposed a new approach *Esh* of calculating binaries’ similarity, which firstly decomposes binary codes into small comparable fragments, then defines semantic similarity between fragments, and further uses statistical reasoning to lift fragment similarity into the similarity between procedures. Chandramohan et al. [39] proposed *Bingo*, which captures complete function semantics by inlining the relevant library and developer-defined functions and models binary functions in a program structure agnostic fashion by using length variant partial traces. Feng et al. [40] proposed *Genius* to search vulnerabilities in massive IoT ecosystems by converting Attributed Control Flow Graph (ACFG) into high-level numeric feature vectors. Different from *Genius* that embeds an ACFG by taking a codebook-based approach, Xu et al. [41] proposed *Gemini* to take a neural network-based approach to transform the ACFGs into embeddings of binary functions for similarity detection. Liu et al. [42] proposed a solution named *αDiff*, which employs three semantic features: intra-function feature, inter-function feature and inter-module feature, to address cross-version binary code similarity detection challenges.

Dynamic Analysis. A dynamic approach for binary code reuse detection is performed during code execution by running programs. Tian et al. [43], [22] proposed *DYKIS* to uniquely identify a program for detecting similar programs. Tian et al. [44] presented a framework called Thread Oblivious Dynamic Birthmark (TOB) that revives existing techniques to detect reuse of multi-thread programs. Ming et al. [45] proposed a logic-based approach *LoPD* by leveraging dynamic symbolic execution and theorem proving techniques to capture dissimilarities between two programs in order to rule out semantically different programs.

For better understanding the differences of the above approaches from ISRD, we compare them in Table I in terms of
and respectively represent satisfying the criteria, par-identical library call invocation check are used to find the
Anchor Recognition
Specifically, in the anchors to significantly accelerate function pair matching.

For "intent search based on anchor recognition" is proposed to recognize anchors and conduct intent search originated from the anchors to significantly accelerate function pair matching. Specifically, in Anchor Recognition, strict instruction match and identical library call invocation check are used to find the anchors. Then, in Intent Search, we originate from the anchor pairs to explore potentially matched function pairs based on their function call relationships.

A. Birthmark Generation

A birthmark is a set of characteristics extracted from a program that reflects its semantic behaviors, which can be used to uniquely identify a program and is resilient to semantics-preserving code transformations.

In the literature, there are many proposed birthmarks with different granularities to represent a program. But in practice, applying coarse granularity may restrict program similarity catching in a precise way. For example, a program-level birthmark is unable to detect partial reuse, since a candidate program often reuses only a part of a target program. Meanwhile, the birthmark similarity at a fine granularity level cannot be used to infer the similarity at a coarse granularity level. For example, given the similarity at instructions, we cannot conclude that the functions of two programs are similar.

To address this problem, we propose a multi-level birthmark model to characterize a program by involving three-level granularities to form a hierarchical birthmark, from the function level, to the basic block level, and finally the instruction level.

On the top of the hierarchical birthmark, to depict the program semantics from the point of a macroscopic view, we first construct the FCG of a program to describe its behavior. Then, we turn our attention to the individual functions in the program by distilling the function semantics with MBP representation. Finally, to further capture the instruction semantics, we perform normalization on the finest-granularity instructions. Consequently, a program can be identified by a three-level birthmark that contains function birthmark, basic block birthmark, and instruction birthmark. Next, We discuss the details of birthmark at each level.

1) FCG Construction: To capture program semantics at the function level, we construct FCG for both the target program and the candidate program. The FCG captures the functionality and objective of a program semantically from its structural information to profile program behaviors [47], [48], [49].

In order to construct FCG of a given program, we first identify the invocation statements (i.e., “call”) from its assembly code to extract callers and callees. Then, the callers and the callees are added into a graph as nodes. In addition, if a function call relation exists between a caller and a callee, an edge is inserted between them in the graph.
2) **MBP Extraction**: To characterize function semantics, Control Flow Graph (CFG) is applied, which contains detailed information of the basic blocks in a function. In a CFG, each node represents a basic block that is a straight-line piece of code without any branch, and each edge represents the control flow relationship among blocks. A CFG is defined as follows.

**Definition 1: Control Flow Graph (CFG)**: A CFG is a directed graph \( G = (V, E) \).
- \( V = \{v_i | 1 \leq i \leq n\} \) denotes the set of basic blocks of a function, where \( v_i \in V \) is the \( i \)th block.
- \( E \subseteq V \times V \) denotes the set of control flows, where \((v_i, v_j) \in E\) indicates that a control flow is from \( v_i \) to \( v_j \).

However, existing approaches [36], [35] that operated at the boundaries of basic blocks are vulnerable to block splitting or merging when different compilation processes are applied. Our solution to this problem is inspired by TRACY [31], which uses *tracelets* – partial traces of an execution to compare function similarity. Concretely, we propose MBPs that are partial straight-line execution paths between branching nodes in a CFG. A branching node is a node that has more than one successor node.

To extract MBPs, we firstly pre-process the CFG by grouping basic blocks into a number of straight-line paths (i.e., replacing edges that connect two blocks with a single out-edge and a single in-edge, respectively). This kind of grouping does not affect the function semantics and is only for the purpose of simplifying the generated MBPs, which is defined as below.

**Definition 2: Minimum Branch Path (MBP)**: Let a path extracted from a CFG be denoted as a node sequence \( p = \langle v_{p_1}, \ldots, v_{p_n} \rangle \). A path \( p \) is a MBP if the following conditions are satisfied:
- \( v_{p_1} \) is an initial node, which has no predecessor or is a branching node.
- \( v_{p_n} \) is a terminal node, which has no successor or is a branching node.
- No other nodes in \( p \) are initial or terminal nodes.

Unlike the fix-lengthed *tracelet* proposed in [31], MBP is variable in length according to the structure of CFG. It has a number of characteristics making it suitable for representing function semantics:

- **Semantics Exhibition**: The combination of basic blocks and control flow in MBP can represent the execution of a function and capture its semantics.
- **Effectiveness and Efficiency**: Trying to gather and analyze all paths in a CFG is clearly infeasible. MBP effectively cuts down the size and the number of paths of CFG by only considering sub-paths between branching nodes. Thus, it can help in speeding up function semantics matching.
- **Structural Variation Resilience**: The absence of branches in MBP implies that it would be less vulnerable than CFG to structural changes caused by block splitting and merging.

![Algorithm 1: MBP Extration from CFG](image)

**Algorithm 1**: MBP Extraction from CFG

*Input*: \( G = (V, E) \) // \( G \) is a CFG.
*Output*: \( P \) // \( P \) is the set of all MBPs.

1. \( P = \emptyset \)
2. for each \( v_i \in V \) do
3. 3. if \( \text{In}(v_i) = 0 \) or \( \text{Out}(v_i) > 1 \) then
4. 4. \( \text{EXTRACT}(v_i) \rightarrow P \)
5. return \( P \)

**Function** \( \text{EXTRACT}(v_i) : \)
7. 7. \( P_i = \emptyset \)
8. 8. for each \( (v_i, v_j) \in E \) do
9. 9. \( v_i, v_j \rightarrow p \)
10. 10. while \( \text{Out}(v_i) = 1 \) do
11. 11. \( v_{k_i} = v_k \rightarrow p \)
12. 12. \( p \rightarrow P_i \)
13. 13. return \( P_i \)

**Algorithm 2**: MBP extraction from CFG

**Fig. 2**: A sample CFG and its extracted MBPs

- **Resilience to Jump Instruction Variations**: Jump instructions are sensitive to obfuscation. MBP is by nature free from jump instructions due to branch omission.

Algorithm 1 shows the steps of extracting MBPs from a given CFG. The output \( P \) denotes the set of extracted MBPs. The functions \( \text{In} \) and \( \text{Out} \) return the in-degree and out-degree of a given node. Specifically, \( P \) is initialized as an empty set. Then, for every node \( v_i \) in \( V \), if \( v_i \) has no direct predecessor nodes or more than one direct successor node, the function \( \text{EXTRACT} \) is invoked to extract MBPs from \( v_i \). Finally, the extracted MBPs are added to \( P \).

Fig. 2 depicts an example of CFG and its extracted MBPs. We observe that the original basic blocks in a control-flow are grouped as execution flows, which are already determined. There are two basic blocks \( BB_1, BB_2 \) that have more than one direct successor basic blocks. Therefore, the extracted MBPs from the CFG are: \((BB_1, BB_2), (BB_1, BB_3, BB_5), (BB_2, BB_4, BB_5), (BB_2, BB_3, BB_5)\).

3) **Instruction Normalization**: Syntax is the most direct birthmark to represent instruction semantics. However, different compilation processes may cause significant differences in the assemblies [31], [43]. To retain the core semantics of instructions and be resilient to obfuscation introduced by cross-compilation, normalization over instructions is applied.

**Key Instruction Extraction**. First, equally treating all kinds of instructions may be defeated by compilation variations. That is because some kinds of instructions are not closely related to semantics and are easily changed across compilation. To solve this problem, we only consider key instructions. Ideally, the key instructions should constitute a small portion of a whole.
execution sequence and must be relatively unique. Through observation, we find that there exist a large number of data-transfer instructions, such as `push` and `pop`, especially `mov`, in almost all MBPs. These instructions can be discarded because they usually facilitate computations rather than belong to a part of MBP logic. Furthermore, they are easily added and deleted compared with other instructions.

Actually, deleting all data-transfer instructions is probably the simplest approach to solve the problem caused by cross-compilation, but it might lead to the loss of data-transfer semantics. Thus, we only keep the first data-transfer instruction when multiple data-transfer instructions appear continuously.

**Instruction Lifting.** As we discussed earlier, the same operation can be expressed in different instructions. To achieve semantics equivalence on instructions, we lift the instructions into high-level operations. For example, two instructions `inc` and `add` can be mapped to one addition operation.

**Operand Removing.** The operands in the instructions are easily changed in the compilation. Even compiling the same source code with the same compilation settings, the operands can be different due to their differences in memory layout. Therefore, we strip the operands from the instructions.

At this point, by using the proposed multi-level birthmark model, we construct the three-level birthmarks for both target and candidate programs, which give a detailed description of the program semantics from coarse granularity to fine granularity.

**B. Reuse Detection**

After **Birthmark Generation**, given the target program and the candidate program, we can capture their similarity relationship based on their birthmarks for reuse detection.

To trade off between coarse granularity and fine granularity, it is reasonable to take the function as a comparison unit. In order to compare functions, we first perform comparison at the instruction level, then combine the comparison results to compute the similarity at the basic block level, and then the function similarity is determined. Finally, the comparison results between functions are aggregated to capture the similarity between the target program and the candidate program.

In addition, the matching relationship at the three levels is presented to interpret the detection results.

However, matching target functions with a large number of candidate functions remains a major bottleneck. The reason is that the scale of function pair-wise comparison increases exponentially with the number of functions.

To tackle this problem, an efficient matching process that significantly accelerates the search for matched function pairs between the target and the candidate programs is proposed. The process contains two main steps, i.e., **Anchor Recognition** and **Intent Search**, to predict which function pairs are likely to match, thereby making the matching process much more efficient by avoiding unnecessary matching. The process first performs strict instruction match and identical library call invocation check to identify matched anchor function pairs, then it explores neighbors of the matched anchors to find new function pairs with a high similarity score under the guidance of function level birthmark. By doing this, we can effectively reduce the number of comparisons before program similarity score calculation to make reuse detection execute swiftly.

1) **Anchor Recognition:** This step attempts to find matched anchor function pairs, which can efficiently instruct later reused function matching. Specifically, we jointly use two ways to recognize the anchors among a huge number of functions in the candidate program.

**Way 1:** Given a function in the target program, we search the functions in the candidate program that meet strict instruction matching. Here, the strict instruction matching indicates that both the numbers of instructions and their sequences in two functions are exactly the same.

**Way 2:** Considering that library call invocations provide an important partial semantics of a function [50], we also look for identical library call invocations that appeared in both the candidate program and the target program.

Note that **Way 1** and **Way 2** operate on the developer-defined functions and library functions, respectively.

Algorithm 2 shows the process of **Anchor Recognition**. The inputs $D_T$ and $L_T$ denote the developer-defined functions and library functions in the target program, respectively. $D_C$ and $L_C$ denote the developer-defined functions and library functions in the candidate program, respectively. The output $Anchor$ denotes a set of matched anchor function pairs. The function GETFUNCINS takes a function as input and returns all its instructions. At line 6, we use Jaccard distance to measure the similarity between the function pairs in terms of **Way 1**, i.e., their similarity in instructions. After that, we obtain the identical library function invocation pairs in both the target program and the candidate program.

2) **Intent Search:** In the intent search, we originate from the matched anchors to discover new matched function pairs. ISRD loops over each recognized anchor and explores its direct

Algorithm 2: Anchor Recognition

**Input:**

$D_T, L_T$ // $D_T, L_T$ denotes the developer-defined functions and library functions in the target program.

$D_C, L_C$ // $D_C, L_C$ denotes the developer-defined functions and library functions in the candidate program.

**Output:**

Anchor // Anchor denotes the set of matched anchor function pairs

```plaintext
Anchor = {}
foreach $d_T \in D_T$ do
  $l_{d_T} = \text{GETFUNCINS}(d_T)$
  foreach $d_C \in D_C$ do
    $l_{d_C} = \text{GETFUNCINS}(d_C)$
    $\text{sim}_{\text{ins}} = \text{SIMINS}(l_{d_T}, l_{d_C})$
    if $\text{sim}_{\text{ins}} == 1$ then
      Anchor += \{$(d_T, d_C)$\}
  endforeach
endforeach
foreach $l_T \in L_T$ do
  foreach $l_C \in L_C$ do
    if $l_C == l_T$ then
      Anchor += \{(l_T, l_C)$
```
Algorithm 3: Intent Search

Input: Anchor // Anchor denotes the matched anchor function pair set. 
\( f_{TB} \) // \( f_{TB} \) denotes the function which has the highest priority in the target program. 
\( FCG_T, FCG_C \) // \( FCG_T, FCG_C \) denote the function level birthmark of the target program and the candidate program.

Output: FF // FF denotes a set of potential matched function pair of \( f_{TB} \), 
\( FF, Pre, Suc = {} \)
1 foreach \((f_{TB}, f_{TB}) \in \text{Anchor} \in FCG_T \) do 
2 \( Pre \leftarrow (f_{TB}, f_{TB}) \in \text{FCG}_C \)
3 foreach \((f_{TB}, f_{TB}) \in \text{Anchor} \in FCG_T \) do 
4 \( Suc \leftarrow (f_{TB}, f_{TB}) \in \text{FCG}_C \)
5 if len(Pre) > 0 and len(Suc) > 0 then 
6 \( FF \leftarrow Pre \cup Suc \)
7 else 
8 \( FF \leftarrow Pre \cup Succ \)
9

neighbors to find new matched function pairs that have a high similarity score. The valid correspondences found in the previous step are propagated to their neighbors. And the new identified matched function pairs are added as new anchors.

Given a matched anchor function pair \( A \) and \( a \), the basic idea of intent search is that the neighboring functions of \( A \) that are connected to \( A \) with an edge in the function level birthmark usually correspond to those of \( a \).

In order to search for new matched function pairs, we use the function invocation relationship to guide our processing. We consider the cues got from the current state of the matched function pairs and the function call relationship in FCG. Instead of exploring completely at random, we set a priority to the functions that have the most number of matched neighbors. Subsequently, the process should firstly operate on the high priority functions in the candidate program. With this process, all matched function pairs between the target program and the candidate program can be identified.

Algorithm 3 shows the process of Intent Search. The input Anchor denotes the set of matched anchor function pairs; \( f_{TB} \) denotes the function that has the highest priority in the target program; \( FCG_T \) and \( FCG_C \) denote the function level birthmarks of the target program and the candidate program, respectively. The output FF denotes a set of potentially matched function pairs of \( f_{TB} \).

Firstly, for each precursor node \( f_{TB} \) of \( f_{TB} \) in \( FCG_T \), if \( (f_{TB}, f_{TB}) \) is in \( \text{Anchor} \), \( (f_{TB}, f_{TB}) \) is added in \( Pre \), where \( f_{TB} \) is the successor node of \( f_{TB} \) (lines 2-3).

Then for each successor node \( f_{TB} \) of \( f_{TB} \) in \( FCG_T \), if \( (f_{TB}, f_{TB}) \) is in \( \text{Anchor} \), \( (f_{TB}, f_{TB}) \) is added in \( Suc \), where \( f_{TB} \) is the precursor node of \( f_{TB} \) (lines 4-5).

Finally, if \( Pre \) and \( Suc \) are not empty, the intersection of them is added into \( FF \), otherwise their union is added into \( FF \) (lines 6-9).

3) Similarity Calculation: After identifying the potentially matched function pairs, we are able to measure the similarities of two functions by calculating the similarity scores based on their basic block level birthmarks - MBP sets. To compute a similarity score for each pair of MBP sets, we first show how to compute the similarity score for a pair of MBPs using Equation (1).

\[
sim(mbp_1, mbp_2) = \frac{2 \times lcs(mbp_1, mbp_2)}{|mbp_1| + |mbp_2|}
\]

We compute the Longest Common Subsequence (LCS) of two MBPs by utilizing the LCS dynamic programming algorithm, denoted as lcs(mbp_1, mbp_2). Then, the similarity score between two MBPs is calculated by taking the length of LCS divided by the average length of two MBPs. By trying each pair of MBPs, we use the collected individual similarity scores to calculate the similarity of two MBP sets \( MBP_1, MBP_2 \) according to the following Equation (2-3).

\[
sim(MBP_1, MBP_2) = \sum_{mbp_1 \in MBP_1} \sum_{mbp_2 \in MBP_2} \frac{|mbp_1| \times \text{MaxScore}}{|mbp_1|}
\]

\[
\text{MaxScore} = \text{Max}(\text{sim}(mbp_1, mbp_2) \in MBP_2)
\]

For each MBP in \( MBP_1 \), the highest similarity score, denoted as MaxScore, is first obtained using Equation (3) in \( MBP_2 \). Then, the similarity score of two MBP sets \( sim(MBP_1, MBP_2) \) is the sum of the highest similarity score of each MBP in \( MBP_1 \) timed by the length of \( mbp_1 \) and divided by a base, which is the total length of all MBPs in \( MBP_1 \). This allows the similarity score to vary between zero and one inclusively.

The similarities between functions are not sufficient to describe the similarity relationship between the target program and the candidate program. We combine the matched function pairs and the function innovation relationship into a graph that is the similar part of the function level birthmarks, which can reconstruct the similarity scene. Consequently, to prove the similarity between the target program and the candidate program, the similar parts between their three-level birthmarks are distilled to serve as the interpretable detection results. The similarity between the target program and candidate program can be measured according to the following Equation (4).

\[
sim(T, C) = \frac{|FF|}{|C|}
\]

The similarity score of the target program and candidate program \( sim(T, C) \) is calculated by using the number of matched function pairs \( FF \) to divide a base, which is the number function of candidate program. The score denotes the percent of reused functions in the candidate program with regard to the target program.

IV. Evaluation

In this section, we first introduce the setup of our experiments. Then, we answer the following six research questions to validate the performance of our approach.
A. Study Setup

1) Evaluation Datasets: ISRD takes the binary code of program pairs as input. To perform a thorough evaluation, we needed ground-truth datasets of which the binary code really share the same code. To this end, we used two datasets, including a dataset that is constructed by ourselves (Dataset-I) and a widely used benchmark dataset Dataset-II that is provided from Bingo [39] and odiff [42].

To test whether ISRD can successfully detect partial reuses, we collected real-world data from open source platforms to construct Dataset-I. We first downloaded 24 open-source projects from open-source platforms (e.g., Github and SourceForge), which fall into different application domains. Then, we selected and labelled a total of 74 real partial reuses as ground truth by using both a manual method and an automatic method. Finally, a dataset containing 24 programs with 74 partial reuses was constructed. We compiled these 24 programs into binaries using gcc with default optimization (O2). The static information of Dataset-I is listed in Table III, where the numbers in the third and the fourth columns are the lines of program source code and the number of functions in the corresponding compiled binaries, respectively. After compiling, some programs would generate more than one binary. Herein, we only present the information of the binaries that are used in our evaluation. We have published Dataset-I at [38].

Dataset-II is a widely-used benchmark to evaluate cross-compilation robustness. It was commonly used in the literatures [39], [42], [51], [52]. Thus, it was adopted to compare the performance of ISRD with existing approaches. The binaries in Dataset-II were compiled from the experimental object Coreutils [53], which are the basic file, shell and text manipulation utilities of the GNU operating system written in C. There are 107 components in Coreutils and as a result, each compilation produces 107 binaries. Following [39], [42], [35], [36], we used three compilers in our experiment (gcc v4.6, gcc v4.8, and clang v3.0) and four optimization levels (O{0, 1, 2, and 3}), resulting in 12 different variants for each of the 107 binaries. Table III lists the information of Dataset-II, where the numbers in the 4th-7th columns denote the maximum, minimum, average, and total numbers of functions of the 107 binaries in Coreutils, respectively.

2) Evaluation Metrics: The metrics used to measure the performance of ISRD are shown in Table IV. False Positive Rate (FPR) stands for the ratio of non-reusable functions being falsely detected as reusable functions. False Negative Rate (FNR) quantifies the ratio of the reusable functions that are not detected as reusable functions. The values of precision, Recall and F-Measure are calculated as described in Table IV.

3) Parameter Setting and Experimental Environment: Similarity threshold plays an important role in ISRD. A function pair is determined as matched if its similarity score is greater than the similarity threshold. To determine a proper threshold, we varied its values to test over both datasets. According to our testing results, setting the similarity threshold as 0.5 made ISRD achieve the best performance.

We implemented ISRD in python 3.6 on Ubuntu 18.04. All programs ran at DELL desktop P2417H with CPU i7-7700 & 3.60GHz and 16GB memory. In the experiments, ISRD utilized angr [78] to disassemble binaries.

B. Answer to RQ 1: Effectiveness and Efficiency of ISRD

In this evaluation, we used Dataset-I to test whether ISRD can effectively and efficiently detect partial reuse. We ran ISRD with Dataset-I as input and compared its results with the titles:

**TABLE II: Descriptions of Dataset I.**

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th># LOC</th>
<th># Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2 [54]</td>
<td></td>
<td>6019</td>
<td>84</td>
</tr>
<tr>
<td>zstd [55]</td>
<td></td>
<td>76465</td>
<td>848</td>
</tr>
<tr>
<td>lzo [56]</td>
<td></td>
<td>78667</td>
<td>895</td>
</tr>
<tr>
<td>lzlib [65]</td>
<td></td>
<td>23736</td>
<td>206</td>
</tr>
<tr>
<td>minizip [57]</td>
<td></td>
<td>24002</td>
<td>208</td>
</tr>
<tr>
<td>precomp [58]</td>
<td></td>
<td>28478</td>
<td>577</td>
</tr>
<tr>
<td>TurboBench [59]</td>
<td></td>
<td>96840</td>
<td>1739</td>
</tr>
<tr>
<td>Dbench [60]</td>
<td>1.8</td>
<td>207315</td>
<td>2664</td>
</tr>
<tr>
<td>brotli [61]</td>
<td>1.07</td>
<td>30072</td>
<td>233</td>
</tr>
<tr>
<td>libbsc [62]</td>
<td>3.10</td>
<td>9192</td>
<td>70</td>
</tr>
<tr>
<td>libdeflate [63]</td>
<td>1.6</td>
<td>79071</td>
<td>70</td>
</tr>
<tr>
<td>lfsre [64]</td>
<td>1.0</td>
<td>3382</td>
<td>35</td>
</tr>
<tr>
<td>lzlib [65]</td>
<td>1.11</td>
<td>4709</td>
<td>98</td>
</tr>
<tr>
<td>zib [66]</td>
<td>1.12</td>
<td>25504</td>
<td>133</td>
</tr>
<tr>
<td>zib-ng [67]</td>
<td></td>
<td>14288</td>
<td>187</td>
</tr>
<tr>
<td>csc [68]</td>
<td></td>
<td>6373</td>
<td>119</td>
</tr>
<tr>
<td>gipfeli [69]</td>
<td></td>
<td>1112</td>
<td>84</td>
</tr>
<tr>
<td>blosc [70]</td>
<td>1.18</td>
<td>58867</td>
<td>742</td>
</tr>
<tr>
<td>libz [71]</td>
<td>1.0.10</td>
<td>1648</td>
<td>30</td>
</tr>
<tr>
<td>Exserver [73]</td>
<td></td>
<td>5625</td>
<td>133</td>
</tr>
<tr>
<td>ckenit [74]</td>
<td></td>
<td>6054</td>
<td>139</td>
</tr>
<tr>
<td>exjson [75]</td>
<td></td>
<td>3384</td>
<td>83</td>
</tr>
<tr>
<td>libdfile [76]</td>
<td>1.0.26</td>
<td>50764</td>
<td>4438</td>
</tr>
<tr>
<td>sndfile2k [77]</td>
<td></td>
<td>59532</td>
<td>39299</td>
</tr>
</tbody>
</table>

**TABLE III: Descriptions of Dataset II.**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Optimization Level</th>
<th>Max</th>
<th>Min</th>
<th>Average</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>v4.6</td>
<td>O0</td>
<td>420</td>
<td>17</td>
<td>151</td>
<td>16183</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O1</td>
<td>297</td>
<td>14</td>
<td>113</td>
<td>12095</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O2</td>
<td>218</td>
<td>14</td>
<td>80</td>
<td>8627</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O3</td>
<td>192</td>
<td>14</td>
<td>68</td>
<td>7365</td>
</tr>
<tr>
<td>clang</td>
<td>v3.0</td>
<td>O0</td>
<td>419</td>
<td>16</td>
<td>150</td>
<td>16076</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O1</td>
<td>279</td>
<td>13</td>
<td>109</td>
<td>11736</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O2</td>
<td>210</td>
<td>13</td>
<td>74</td>
<td>7954</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O3</td>
<td>184</td>
<td>13</td>
<td>67</td>
<td>7243</td>
</tr>
</tbody>
</table>

In this evaluation, we used Dataset-I to test whether ISRD can effectively and efficiently detect partial reuse. We ran ISRD with Dataset-I as input and compared its results with the titles:
TABLE IV: Evaluation Metrics.

<table>
<thead>
<tr>
<th>Term</th>
<th>Abbr</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>TP</td>
<td>the number of reusable functions that are correctly detected as reusable.</td>
</tr>
<tr>
<td>True Negative</td>
<td>TN</td>
<td>the number of reusable functions that are correctly detected as reusable.</td>
</tr>
<tr>
<td>False Negative</td>
<td>FN</td>
<td>the number of reusable functions that are incorrectly detected as reusable.</td>
</tr>
<tr>
<td>False Positive</td>
<td>FP</td>
<td>the number of reusable functions that are incorrectly detected as reusable.</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>FPR</td>
<td>$FP/(FP + TN)$</td>
</tr>
<tr>
<td>False Negative Rate</td>
<td>FNR</td>
<td>$FN/(TP + FN)$</td>
</tr>
<tr>
<td>Precision</td>
<td>P</td>
<td>$TP/(TP + FP)$</td>
</tr>
<tr>
<td>Recall</td>
<td>R</td>
<td>$TP/(TP + FN)$</td>
</tr>
<tr>
<td>F-measure</td>
<td>$F_1$</td>
<td>$2PR/(P + R)$</td>
</tr>
</tbody>
</table>

![CDF of Precision](image1.png) ![CDF of Recall](image2.png)

**Fig. 3: CDFs of ISRD Precision and Recall on Dataset I**

ground truth. Fig. 3 and Fig. 4 report the performance of ISRD in terms of effectiveness and efficiency to detect partial reuse.

Fig. 3 presents the Cumulative Distribution Function (CDF) for the precision and recall of the evaluation results. Almost all the precision values are higher than 82% and its average value can reach 97.2%. Moreover, 15 binary pairs achieve precision equal to 1, indicating that all reused functions can be accurately detected as reusable. For the recall metric, its average value reaches 94.8%, indicating that our approach can effectively find reused function pairs.

Fig. 4 reports the CDF with regard to the ratio of the reduced number of function pairs that were calculated by ISRD to the original number of function pairs. We can observe that ISRD effectively reduces the size of the function pairs by 97.9% on average during reused function detection. Specifically, when the reused code is only a small fraction of the target program and the candidate program, the reduced ratio is up to 99.9%. For example, the reused code of bzip2 only account for about 2% of precomp, 208824 functions pairs are needed to be calculated if applying some existing approaches (i.e., BinDiff [34], TRACY [31]), whereas 128 function pairs are calculated by ISRD. Thus, we can conclude that the number of function pairs used in ISRD is much smaller than the total number of all function pairs between the target program and the candidate program. This huge reduction in reuse detection makes ISRD practical for large real-world programs.

**Answering RQ 1: ISRD effectively detect partial reuse and its average precision achieves 97.2%. ISRD can significantly reduce the number of function pairs required for reuse detection up to 97.9% on average. Thus, it is efficient to handle a large scale of programs.**

**C. Answer to RQ 2: Interpretability of ISRD**

To evaluate whether the detection results of ISRD are interpretable, we leveraged two programs precomp [58] and minizip [57] in Dataset I to perform a case study. The case study illustrates the interpretability of ISRD by providing a graphic demonstration of reuse. Herein, precomp is a command line precompressor and minizip is a zip manipulation library in C. Note that both precomp and minizip reuse two compression libraries lzma [79] and bzip2 [54].

Fig. 5 illustrates the detection results of ISRD. The FCGs of precomp and minizip are presented on the left side of the figure, where the nodes in red and in blue represent the functions in the library lzma and bzip2, respectively. Running ISRD on precomp and minizip, we got program similarity as 17.9%, which implied that 17.9% functions of precomp have been already used in minizip.

The detailed detection results are illustrated and interpreted on the right side of the figure. For presentation purpose, only the lzma is given. After identifying all matched function pairs between precomp and minizip, the matched subgraph were constructed by combining the matched function pairs with their function call relationship in order to reconstruct the reuse scene in the function level of the two programs. The matched subgraph between the two programs is presented in Fig. 5 (1). In the matched subgraph, the node pairs are the matched function pairs between the two programs that have the same function call relationship.

Furthermore, we singled out function lzma alone decoder with its matched function, whose similarity score is 1.0, as an example by including the matched part of this function at the basic block level and the instruction level. In the basic block level, as showed in Fig. 5 (2), 4 matched MBP pairs were recognized to demonstrate the similarity between the function pair. The matched MBPs are displayed with the same colors.

Finally, the similarity relationship of the last matched MBP pair is described in detail at the instruction level. 9 high-level matched operation pairs explain the semantic equivalence at the finest granularity, as displayed in Fig. 5 (3) with a similarity score as 1.0.

**Answering RQ 2: The case study shows that the detection results of ISRD is interpretable by describing the matched part between the target program and the candidate program in detail at the function level, the basic block level and the instruction level.**
**D. Answer to RQ 3: Resilience to Cross-Compiler-Version**

We compiled *Coreutils* using *gcc* v4.6 and *gcc* v4.8 with various optimization levels (O0 to O3). Such a setup led to 8 different versions for each binary in *Coreutils*. Subsequently, we evaluated ISRD by comparing the binaries compiled using different compiler versions with the same optimization levels.

Table V summarizes this experiment’s results, where each row heading represents the optimization level used for compilation, and the last row reports the average value of each evaluation metric. For example, the second row represents the detection results when the target programs and the candidate programs were compiled using *gcc* v4.8 with O0 while the candidate programs were compiled with O1 using the same compiler.

The results demonstrate that the average precision can achieve 96.0% and the recall is 99.3%, indicating that ISRD is resilient to the obfuscation caused by different compiler versions. Moreover, for no code optimization (i.e., O0), the precision is 100% while the FPR is 0%. That is, even with different version compilers, the compilations without code optimization levels lead to highly similar binary codes, whereas compiling with high optimization levels yields more differences between binaries.

**Answering RQ 3: ISRD is resilient to cross-compiler-version with 96.0% precision and 99.3% recall on average.**

**E. Answer to RQ 4: Resilience to Cross-Optimization-Level**

We compiled *Coreutils* for x86-32bit architecture using *clang* v3.0 and *gcc* v4.8 with various optimization levels (O0 to O3). With this setup, each binary in *Coreutils* had 8 variants. We compared the binaries compiled using the same compiler with different optimization levels.

Table VI summarizes the results, where each row heading represents the optimization level used to compile the detection objects. For example, the second row represents the detection results of the target programs that were compiled by *gcc* v4.8 with O0 while the candidate programs were compiled with O1 using the same compiler.

We observe that ISRD performs much better regarding precision in *clang* v3.0 than in *gcc* v4.8. Specifically, the average precision is 95.7% for *clang* v3.0 yet only 91.1% is obtained for *gcc* v4.8.

Another interesting observation is that we get similar patterns for both compilers. Within one compiler type (e.g., *gcc* v4.8), the F-measure achieved by matching between the binaries that were all compiled with high code optimization levels (i.e., O2 and O3) is always better than that obtained by matching between the binaries where one is compiled with a high code optimization level (i.e., O2 and O3) and the other is compiled with no code optimization (i.e., O0). Specifically, the highest F-measure is achieved when the target program and the candidate program were both compiled with high optimization, O2 and O3, respectively. However, F-measure drops to 72.0% if the target program and the candidate program were respectively compiled with no code optimization (i.e., O0)
and high code optimization level (i.e., O3). This suggests that regardless of the compiler vendor, the binaries compiled with high optimization levels are similar. The influence of whether the optimization kicks into the binaries is very evident.

**Answering RQ 4:** ISRD is resilient to cross-optimization-level to some extent. Evaluating the binaries compiled by gcc v4.8 and clang v3.0, ISRD achieves on average 91.1% precision and 95.7% precision, respectively.

**F. Answer to RQ 5: Resilience to Cross-Compiler-Vendor**

We compiled Coreutils for x86-32bit architecture using clang v3.0 and gcc v4.8 with various optimization levels (O0 to O3). 8 different variants were generated for each binary in Coreutils with this setup. Subsequently, we evaluated ISRD on binaries compiled by different vendors’ compilers with same optimization levels.

We list the results in Table VII, where each row heading represents the optimization level used to compile the compared objects. Experimental results show that the average precision reaches a high degree, 94.2% with FPR as 0.1%. From the table, we find that the best results are obtained when the candidate programs were compiled with no optimization level as the target one. Besides, the precision hits the lowest point of 91.2% when the detection binaries were compiled with optimization level O3. Further, across compiler vendors, the detection precision drops with the rise of the optimization level, except when the detection programs were compiled with optimization level O1, where the detection programs compiled with optimization level O2 yield better accuracy.

**Answering RQ 5:** Experimental results demonstrate the resilience of ISRD across compiler vendors. Moreover, it achieves on average 94.2% precision.

**G. Answer to RQ 6: Comparison with Related Works**

In this experiment, we compared ISRD with three baseline approaches, including, BinDiff [34], BinGo [39], and αDiff [42].

1) **BinDiff** [34] is a binary code similarity detection tool, which matches a pair of binaries using a variant of graph-isomorphism algorithm.

2) **Bingo** [39] is a scalable and robust binary search engine that supports cross-compilation by applying a selective inlining technique to recover complete function semantics.

3) **αDiff** [42] is a method to detect binary code similarity with a neural network solution to extract intra-function semantic features from raw bytes of binary functions.

To make a head-to-head comparison with these approaches, we used the same experimental configurations and the same metric as theirs. More specifically, we compiled Coreutils using gcc v4.8 and clang v3.0 with various optimization levels (from O0 to O3). We evaluated six experimental settings. The recall values of reuse detection are reported in Table VIII, where each row heading represents the optimization level used to compile the target programs and the candidate programs.

Through comparison on recall, we can see that ISRD outperforms the three baseline approaches by 27.0% on average, especially outperforms BinGo by 36.5% on average. We note that the average recall of ISRD is 76.4%, which is lower than the upper experiment results. This is because the obfuscations of both cross-compiler-vendor and cross-optimization-level were involved, making the detection much harder than the situation where only one type of obfuscation occurs. Moreover, the worst result was obtained by each approach when the target programs were compiled by gcc v4.8 with no code optimization and the candidate programs were compiled by gcc v4.8 with the highest optimization level O3. On the other hand, the best results of all approaches were achieved when the target programs and the candidate programs were compiled by the same compiler with the high optimization level, O2 and O3, respectively.

**Answering RQ 6:** Compared with the baseline approaches, ISRD achieves better performance regarding a widely used benchmark dataset.

**V. LIMITATIONS OF ISRD**

**A. Impacts Caused by Internal Reasons**

**Function Inlining.** Function inlining is a major internal limitation to ISRD. In ISRD, the function semantics inside a program is distilled independently. To be specific, a callee function’s semantics is not integrated into a caller function’s semantics. This leads to a partial semantics problem, because the strategy of inlining is selectively applied according to a configured optimization level during compilation. Inlining is utilized in compilers to optimize the binaries for achieving maximum speed or minimum size [80]. Other approaches (e.g., the selective inlining strategy of Bingo [39]) that inline relevant libraries and developer-defined function semantics can be incorporated into ISRD to address the problem.
Library Functions. In ISRD, two ways are leveraged to recognize the anchors. The second way is based on checking identical library call invocations. However, this way is limited for two reasons: (1) the library functions are OS dependent; (2) it fails to recognize the library calls that have different names yet with similar functionality (e.g., memcpy and memmove) [39]. To address the above problems, inspired by CLCDSA [81], the similarity of cross-os library calls can be learned with the help of the documentation and Mikolov’s Word2Vec [82] model.

B. Impacts Related to Datasets

Due to the difficulty of collecting partial reuse samples with accurate labels, only 24 programs were selected to construct Dataset I and 74 real partial reuses were manually labeled. In future work, we plan to collect additional real-world partial reuse samples and further evaluate and optimize the performance of ISRD.

VI. CONCLUSION

In this paper, we proposed ISRD, an interpretation-enabled software reuse detection approach based on a multi-level birthmark model. We used the multi-level birthmark model to distill the program semantics from coarse granularity to fine granularity. Through normalization, the function semantics and the core semantics of instructions can be effectively represented. In reuse detection, intent search originated from recognized anchors was employed to speed up function pair matching. Extensive experiments reveal that ISRD is effective and efficient in detecting partial reuse with interpretation. It also outperforms state-of-the-art approaches in terms of resilience to cross-compilation.

ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China (2018YFB1004500), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61532015, 61833015, 61902306, 62072351), China Post-doctoral Science Foundation (2019TQ0251, 2020M673439), Youth Talent Support Plan of Xi’an Association for Science and Technology (095920201303), Ministry of Education Innovation Research Team (IRT_17R86), and Project of China Knowledge Centre for Engineering Science and Technology, as well as Academy of Finland (Grants 308087 and 335262).

REFERENCES


