SBridge: Identifying Source-to-Binary Function Similarity via Cross-Domain Control Block Matching

HEEDONG YANG, Korea University, Republic of Korea
JEONGWOO LEE, Korea University, Republic of Korea
HAJIN YUN, Korea University, Republic of Korea
SEUNGHOON WOO*, Korea University, Republic of Korea

We present SBridge, a precise approach for identifying functions in binaries that are similar to the given source code functions. Identifying reused code in binaries is critical for security, particularly for detecting propagated vulnerabilities.Although binary-to-binary comparison is feasible, leveraging source code as the reference is more practical because source code is easier to collect and analyze directly without compilation.However, significant gaps between source and binary representations, including function inlining, create challenges in cross-domain function detection.Existing approaches primarily rely on string literals or structural similarities between entire functions, failing to capture detailed code behavior and generating many false alarms.

SBridge addresses these limitations through a key innovation: control block-based function matching, which encapsulates essential functional features by segmenting functions into meaningful units such as conditionals and loops. Leveraging control blocks as a cross-domain representation, SBridge enables precise measurement of function similarity between source and binary code, effectively overcoming challenges posed by function inlining and stripped binaries. For evaluation, we collected 3,904 real-world C/C++ binaries from BinKit. In experiments identifying binary functions identical to input source functions, despite approximately 40% of binary functions being inlined, SBridge achieved 75.13% *recall@1* and 80.98% *recall@5*, outperforming existing approaches, which achieved up to 43.31% *recall@1* and 50.2% *recall@5*. Our further analysis confirmed that SBridge effectively identifies propagated vulnerabilities in binaries.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Source-to-Binary Matching, Clone Detection, Vulnerability Management.

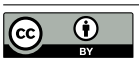
ACM Reference Format:
Heedong Yang, Jeongwoo Lee, Hajin Yun, and Seunghoon Woo. 2026. SBridge: Identifying Source-to-Binary Function Similarity via Cross-Domain Control Block Matching. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE062 (July 2026), 23 pages. https://doi.org/10.1145/3797090

1 Introduction

Open-source software (OSS) has become ubiquitous in software development [32, 38]. However, this widespread adoption has introduced new challenges in software security [23, 42]. In particular, as Commercial-off-the-shelf (COTS) binaries increasingly incorporate OSS components, reused code analysis has become essential for vulnerability prevention and license compliance [10, 32, 37].

*Corresponding author

Authors' Contact Information: Heedong Yang, Korea University, Seoul, Republic of Korea, heedongy@korea.ac.kr; Jeongwoo Lee, Korea University, Seoul, Republic of Korea, jeongwoo@korea.ac.kr; Hajin Yun, Korea University, Seoul, Republic of Korea, hajinyun1011@korea.ac.kr; Seunghoon Woo, Korea University, Seoul, Republic of Korea, seunghoonwoo@korea.ac.kr.



ACM 2994-970X/2026/7-ARTFSE062
https://doi.org/10.1145/3797090

Therefore, *code similarity measurement* has emerged as a key technique for enabling software composition analysis (SCA) to manage OSS components in COTS binaries [9, 11, 21, 30, 41] and detect propagated vulnerabilities [38–40, 55]. Traditionally, *binary-to-binary* matching is dominated by comparing target binaries against binary code databases (*e.g.*, [24, 33, 45, 49]). However, this approach is impractical for real-world use because maintaining a database of numerous compiled software variants is costly and time-consuming.

To address this issue, *source-to-binary* matching emerged (*e.g.*, [10, 11, 20]), leveraging source code databases and focusing on compilation-resilient features (*e.g.*, string literals).

Although source-to-binary matching effectively identifies OSS, existing efforts typically operate at a coarse granularity, lacking precision for detailed function-level analysis. A more refined approach could enable precise detection of binary code fragments (*e.g.*, functions) that closely correspond to source inputs, thereby enhancing both binary SCA and vulnerability detection.

However, achieving precise source-to-binary matching faces the following three fundamental challenges (details are introduced in Section 2.1).

(1) **Loss of source code context.** The compilation process strips crucial information (*e.g.*, variable names), widening the gap between source and binary representations.

(2) **Architecture and compiler variability.** Binary code structure and behavior vary significantly across different hardware architectures and compiler configurations.

(3) **Addressing function inlining.** Modern compilers optimize performance by inlining functions, embedding callee functions' code at call sites.

Limitations of existing approaches. Existing source-to-binary matching approaches (*e.g.*, [10, 11, 20]) have focused on coarse-grained OSS detection. Therefore, they are ineffective for function similarity measurement, especially due to the aforementioned challenges. For example, B2SFinder [11] relies on features susceptible to compiler-induced variations (*e.g.*, control flow), reducing efficiency in source-binary matching. Meanwhile, the state-of-the-art approach BinaryAI [20] overlooks function inlining, leading to low accuracy in identifying binary functions similar to source functions (see Section 4.1). Alternatively, a source-to-source matching strategy (*e.g.*, [31, 36, 53]) can also be considered, where *decompiled* binaries are compared with source code. However, decompiled code often deviates significantly from the original source in syntax, leading to low detection accuracy. Moreover, such approaches are particularly ineffective when function inlining has occurred.

To overcome their shortcomings, we present SBridge (Source-to-Binary bridge), a precise approach for identifying functions in binaries that are similar to the input source functions.

Our approach. The key technical contribution of SBridge is a *control block-based function similarity measurement* technique, which introduces a new granularity called control blocks to capture essential structural and semantic features for accurate source-to-binary matching.

Given input source functions and target binary, SBridge first extracts *control blocks*—fine-grained units encapsulating structural and semantic features (*e.g.*, `if` condition blocks)—from both the source and decompiled binary functions (see Section 3.1). The core insight is that control blocks preserve their functional integrity even after function inlining, enabling SBridge to identify cross-domain function similarities. SBridge considers both internal and external branching patterns of functions, and categorizes control blocks into seven types (see Table 1). It then extracts *key features* essential for matching (*e.g.*, condition expressions) and *block contents* used as auxiliary information (*e.g.*, syntax of code lines within conditional statements) from each block (see Section 3.1).

SBridge computes function similarity based on the similarity of their constituent blocks. Using the source function's blocks as reference points, it quantifies the ratio of these blocks contained within a binary function, thereby preserving block-level similarity even when multiple source

functions are inlined into a binary function. In addition, a weighting mechanism based on call relationships and function length is applied to further address function inlining (see Section 3.2). Block similarity is computed from key features and block content, and function similarity is finally derived from the proportion of matching blocks between two functions (see Section 3.3). Consequently, SBridge can identify, for each input source function, the corresponding similar functions within the target binary.

Evaluation. For our experiments, we collected 3,904 real-world C/C++ binaries from BinKit. Using the source code of each binary as input, we evaluated the ability to locate compiled functions from their source counterparts. When compiling with O2 optimization, we observed that approximately 40% of source functions underwent function inlining during compilation (see Section 4.2). Nevertheless, SBridge achieved 75.13% `recall@1` and 80.98% `recall@5`, significantly outperforming existing approaches [20, 53] (up to 43.31% `recall@1` and 50.2% `recall@5`; see Section 4.1). Further experiments on SBridge's effectiveness and performance demonstrated that SBridge (1) robustly maps source and binary functions even in the presence of function inlining (see Section 4.2), (2) detects functions within an average of 1.9 s per binary, proving its practicality (see Section 4.3), and (3) effectively identifies propagated vulnerabilities in binaries (see Section 4.4).

Contributions. This paper makes the following three contributions.

- We present SBridge, an approach to effectively detect binary functions similar to given source functions. Its key contribution is a control block-based function matching technique.
- SBridge outperforms existing approaches by achieving a `recall@1` of 75.13% and a `recall@5` of 80.98% in mapping source functions to their compiled binary counterparts.
- By introducing an effective source-to-binary function mapping method, SBridge establishes a foundation for future advancements in static analysis of source and binary code.

2 Motivation

In this section, we clarify the target problem addressed by SBridge and explore its motivation through a motivating example.

2.1 Problem and Technical Challenges

We focus on the problem of precisely detecting similar code by comparing source code and binary. Specifically, given a set of source code functions, SBridge attempts to identify and map a set of binary functions within a target binary that exhibit high similarity to the input source functions.

However, this *source-to-binary* matching problem is particularly complex owing to the transformative nature of the compilation process and the diversity of execution environments. Hence, to effectively perform source-to-binary matching, the following three technical challenges should be addressed (see Figure 1).

C1: Loss of source code context. Many details contained in the source code are lost during compilation into binaries or when converting it into more human-readable low-level pseudocode (*e.g.*, decompiled code). For example, variable names, data types, and the structure of the code are not preserved in the same way from source code to binaries. In particular, the distribution of most COTS software is as stripped binaries with static symbol tables (*e.g.*, `.symtab` and `.strtab` sections) removed for security or licensing reasons. This semantic gap presents a fundamental challenge in making meaningful comparisons between source and binary functions.

C2: Architecture and compiler variability. Different operating systems and processor architectures significantly impact how code is executed and represented in binary, including variations in system calls and library implementations. Compiler selection, versions, and optimization settings

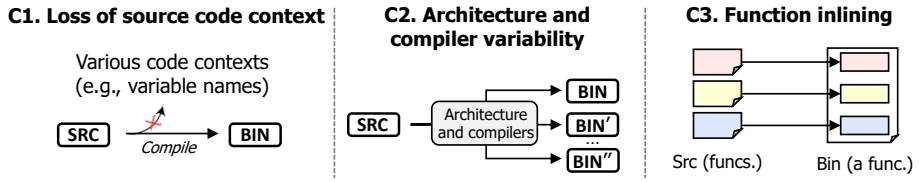


Fig. 1. Three major challenges in matching source code and binaries.

further change the compiled output, making it difficult to match binaries to their original source. Even with the same source function, different compilers may use different standard C/C++ libraries based on optimization options. Addressing this diversity in architecture and compilers remains a key challenge. The diversity in architecture and compilers also represents a significant problem that needs to be addressed.

C3: Addressing function inlining. Function inlining poses a major challenge as it follows complex, unpredictable patterns, often applied for performance optimization or reducing function call overhead. In binaries, multiple source functions may be merged into a single binary function (see Section 2.2), complicating similarity analysis between source and binary functions. Despite existing efforts to tackle function inlining, achieving a robust mapping from source to binary functions continues to be a significant challenge [17–19].

2.2 Motivating Example

To motivate our goal, we examined the `csplit` binary from GNU Coreutils 9.0, especially focusing on the `main`, `parse_patterns`, and `extract_regexp` functions. The binary was compiled for the ARM 32-bit architecture using Clang 13.0 with `-O2` optimization and stripped of symbols. This example highlights how the selected challenges, particularly function inlining, manifest in practice.

Listing 1, Listing 2, and Listing 3 present source code snippets of three selected functions, while Listing 4 shows a decompiled code snippet of the `main` function from the `csplit` binary.

A key observation is *function inlining*: in the source code, `main` calls `parse_patterns`, which in turn calls `extract_regexp`. However, both the `parse_patterns` and `extract_regexp` functions are inlined into `main` in the target binary. Moreover, compilation causes substantial information loss; for example, variable and parameter names are removed and altered, literal values are modified (*e.g.*, `'f'` → `0x66`), and function call statements disappear due to inlining (*e.g.*, line #8 in Listing 1).

Existing approaches. Function inlining, together with information loss during compilation, significantly undermines the effectiveness of existing source-to-binary matching approaches (*e.g.*, [11, 20]). For example, they would attempt to find a source function corresponding to `FUN_000115f4`, but may only match one of the three original functions or fail to identify a corresponding source function. Similarly, source-to-binary matching becomes challenging, as the syntax of `FUN_000115f4` differs significantly from the original source functions. These tendencies were most prominently observed in the evaluation, manifesting as limitations of existing techniques (see Section 4.1).

SBridge. SBridge aims to identify binary functions similar to input source functions, particularly in cases where decompiled code deviates significantly from its source. The approach centers on extracting control blocks from each function. For example, the `switch-case` statement in Listing 1 is extracted as a `Condition` block, while the call to `parse_patterns` is identified as a `CalleeFunction` block (see Section 3.1). SBridge's block matching approach, designed to recognize code changes in block structures, demonstrates that even when the original `CalleeFunction` block from the source `main` function is not found in the decompiled code, the presence of various blocks

Listing 1. The `main` function of `csplit`.

```
int main (int argc, char **argv)
{ ...
  switch (optc) {
    case 'f':
      prefix = optarg;
      break;
  ...
  parse_patterns (
    argc, optind, argv);
```

Listing 2. The `parse_patterns` function of `csplit`.

```
static void parse_patterns (
  int argc, int start, char *argv) {
  ...
  if (*argv[i] == '/' ||
    *argv[i] == '%')
    p = extract_regexp (i,
    *argv[i] == '%', argv[i]);
  ...
```

Listing 3. The `extract_regexp` function of `csplit`.

```
static struct control *
extract_regexp (int argnum,
  bool ignore, char const *str)
{ ...
  char delim = *str;
  char const *closing_delim;
  closing_delim =
    strrchr (str + 1, delim);
  if (closing_delim == NULL)
  ...
```

Listing 4. Decompiled code of `main` (`FUN_000115f4` in the stripped binary) extracted from the `csplit` binary.

```
undefined4 FUN_000115f4 (int param_1, undefined4 *param_2) {
  ...
  switch(iVar7) {
    case 0x66:
      *piVar17 = *DAT_00012744;
      break;
  ...
  if (uVar28 == 0x2f || uVar28 == 0x25) {
    pcVar11 = strrchr((char *)(pbVar24 + 1), uVar28);
    if (pcVar11 == (char *)0x0)
  ...
```

(including `Condition` blocks and those omitted in Listing 1) within the decompiled function indicates that `main` maintains high similarity with its decompiled counterpart. Similarly, SBridge identifies that the remaining two functions also exhibit high similarity to the `FUN_000115f4` function. Notably, SBridge's weighting mechanism based on call relationships and function length helps better account for inlining effects during similarity measurement (see Section 3.3).

3 Design of SBridge

In this section, we describe the design of SBridge, an approach for precisely detecting functions in a target binary that are similar to given source functions.

Overview. SBridge utilizes a new unit called a *control block* to identify binary functions similar to source functions (see Section 3.1). The brief definition of a control block is as follows.

- **Definition: Control block**

  A control block is a fundamental code unit within a function that encapsulates key functional features and represents distinct structural components, such as loops, if-else blocks, and embedded string elements.

Figure 3 depicts the workflow of SBridge, which comprises three main phases: *control block extraction* (**P1**), *control block similarity measurement* (**P2**) and *block-based function matching* (**P3**).

In **P1**, given the source code and binary, SBridge first extracts functions and then constructs control blocks for each function. To bridge the gap between source and binary representations, SBridge performs normalization, reducing differences between them and transforming them into a comparable format. In **P2**, SBridge computes the similarity between control blocks. To this end, it extracts *key features* that characterize each block and *block contents* as auxiliary information for matching. Using these elements, SBridge performs block-level matching, identifying similar control block pairs among all control block combinations between two functions. In **P3**, SBridge measures function similarity based on the block-level similarity scores. It then ranks candidate binary functions according to their similarity to the input source function.

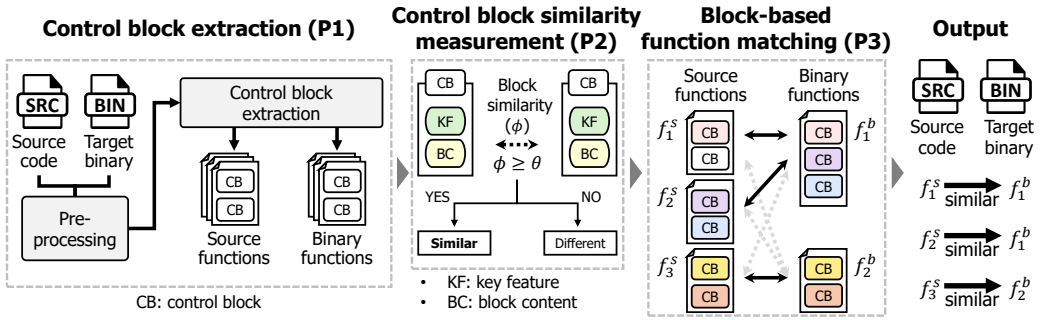


Fig. 3. Overview of SBridge.

Scope and assumption. SBridge operates regardless of whether the binary is stripped. Because of function inlining, a single source function may correspond to multiple binary functions (1-to-N matching), and conversely, multiple source functions may map to a single binary function (N-to-1 matching). Instead of identifying only the most similar binary function for a given source function, SBridge extracts all similar binary functions based on similarity scores, focusing on the top five.

3.1 Control Block Extraction (P1)

Given the source code and target binary, SBridge first identifies the functions and subsequently extracts control blocks from each function.

Preprocessing of source code. Given a C/C++ source code file, SBridge first extracts functions. However, limiting analysis to functions alone may result in the loss of crucial context, such as macro definitions. To preserve this information SBridge preprocesses the source code using the `-E` option (*e.g.*, `gcc -E source.c`). While this does not perform full compilation, it ensures that macro expansions and header file inclusions are reflected in the extracted function bodies. SBridge then uses function parsers (*e.g.*, Ctags [6]) to extract function bodies from the preprocessed source code.

Preprocessing of binary code. SBridge extracts functions from binaries by leveraging a binary reverse engineering tool (*e.g.*, IDA Pro [15], Ghidra [2], Binary Ninja [1] and rev.ng [8]), which can generate C-like low-level pseudocode for all functions within a binary. We use C-like low-level pseudocode instead of raw assembly code to more clearly identify control structures, such as loops and conditionals. This makes it easier to extract control blocks using C/C++ parsers.

Control block extraction. SBridge slices a function into finer-grained units called a *control block*, a minimal semantic unit within a function that encapsulates functional features related to program flow control. The underlying concept is that even when inter-function or intra-function control flow changes, the individual small units remain fundamentally stable. To implement this approach, SBridge considers two branching patterns for control block selection: (1) *internal* and (2) *external*.

- **Internal branching.** This includes conditions, else-conditions, and loop code statements. Although their order and syntax may vary during compilation and decompilation, individual condition units generally preserve their syntactic structure (see Section 2.2).
- **External branching.** This encompasses branching information across functions. It focuses on function call-related constructs, which we further subdivided into three distinct categories: user-defined function calls, C library invocations, and recursive function calls.

Table 1. Types and representative components of defined control blocks.

IDX	Block type	Key feature	Block content
C1	`Condition`	Condition expression	Codes in conditional state
C2	`Else-Condition`	None	Codes in conditional state
C3	`Loop`	Condition expression	Codes in loop state
C4	`String`	String literal	None
C5	`LibcFunction`	(Func. name, #Params)	Parameter information
C6	`CalleeFunction`	(Func. name, #Params)	Parameter information
C7	`RecurFunction`	(Func. name, #Params)	Parameter information

Along with branching blocks, SBridge also considers string literals that are resilient to compilation changes. Based on this classification, SBridge categorizes control blocks into seven types.

C1. `Condition`: Includes conditional expressions (*e.g.*, `if` and `switch` conditions).

C2. `Else-Condition`: Includes alternative branches (`else` conditions).

C3. `Loop`: Includes loop structures (*e.g.*, `for` and `while`).

C4. `String`: Includes string literals, which mostly remain invariant across compilations.

C5. `LibcFunction`: Includes standardized C library function calls.

C6. `CalleeFunction`: Includes user-defined function calls.

C7. `RecurFunction`: Includes recursive function calls.

Each block consists of *key features* and *block contents*.

- **Key feature**: Elements that include the primary feature for comparing blocks.
- **Block content**: Elements that include an auxiliary feature for block matching.

Table 1 summarizes the key features and block contents of each block type. Control blocks are selected based on certain control structures that preserve their logical intent despite syntactic transformations during compilation. `Condition`, `Else-Condition`, and `loop` blocks (C1 to C3) generally maintain the structure of original condition expressions despite potential variations in code syntax due to compilation. The key feature of these blocks is the condition expression, while all code lines contained within each conditional block are stored as the block contents. SBridge extracts control blocks hierarchically to handle nested structures. In a nested conditional statement, the outer block has its own condition as the key feature, which may be empty if it only contains another control block. The inner block is extracted separately with its own key feature and block content.

`String` blocks (C4) contain string literals. Although string literals do not strictly belong to control flow, we include them as an auxiliary type of control block because string literals tend to remain relatively unchanged across compilations.

Next, although the `LibcFunction` (C5) and `CalleeFunction` blocks (C6) can be simplified during optimization, they still encapsulate critical function call information. To extract `LibcFunction` blocks, we selected 346 standard C library functions (*e.g.*, `printf`, `scanf`) based on the IBM Standard C Library Functions Table document, with additional inclusion of GNU gettext (i18n/l10n) and POSIX/BSD socket functions [16, 27]. However, C library functions can be optimized or replaced by alternative functions due to preprocessing options and compiler optimization. Specifically, the LLVM compiler often performs optimizations that replace certain C library functions with simpler equivalents when specific conditions are met, as implemented in the LLVM Library Calls Simplifier

Listing 5. Example code for extracting control blocks.

```c
// KF: Key feature, BC: Block content
#include <stdlib.h>

int add_one(int x) { return x + 1; }
int factorial(int n);
int main(void) {
  int i = 0, p = 2, q = 0, r = 3;
  if (p <= 1) return 0;          // [C1] Condition (KF, BC)
  else { r = r + 5; }            // [C2] Else-Condition (BC)
  while (i < r) {                // [C3] Loop (KF)
    q = q + 1;                   /* [C3] Loop (BC)
    i++;                            [C3] Loop (BC) */
  }
  const char *s = "hi";          // [C4] String (KF)
  void *buf = malloc(16);        // [C5] LibcFunction I (KF, BC)
  free(buf);                     // [C5] LibcFunction II (KF, BC)
  int t = add_one(q);            // [C6] CalleeFunction I (KF, BC)
  int u = factorial(3);          // [C6] CalleeFunction II (KF, BC)
  return t + q + u + (int)s[0];
}

int factorial(int n) {
  if (n <= 0) return 0;          // [C1] Condition (KF, BC)
  return factorial(n - 1);       // [C7] RecurFunction (KF, BC)
}
```

(`SimplifyLibCalls.cpp`). For example, when `printf` is used without format specifiers and the format string ends with a newline character, it can be replaced with `puts` (*e.g.*, `printf("foo\n")` → `puts("foo")`). Because the optimization level of binary code is indeterminate, it is impossible to predict the level of optimization applied. Therefore, by consulting the relevant documentation, we grouped together functions that can be substituted, such as those described in LLVM `SimplifyLibCalls` and the GNU C Library manual [26, 27]. Table 2 shows the representative C library function groups.

Finally, `RecurFunction` blocks (C7) are not affected by function inlining and are also considered by SBridge as a type of control block.

When we examined the proportion of lines belonging to control blocks across all collected source and binary functions in our experimental setup (see Section 4.1), we observed that more than 82.66% of source lines and over 73.77% of binary lines could be mapped to block types. Code lines

Table 2. C library function group (referring to [26, 27]).

Representative function	Group
printf	printf, iprintf, puts, putchar
fprintf	fprintf, fputc, fputs, fwrite, fiprintf, __small_fprintf
strncat	strncat, strcat
strrchr	strrchr, strchr, strlen, strpbrk
strncmp	strcmp, memcmp, strncmp, strstr
memcopy	memcpy, bcopy, memset, strncpy, stpncpy, bcmp, memccpy
gettext	gettext, dgettext, dcgettext, ngettext, dngettext, dcngettext
errno	errno, __errno_location

Table 3. List of features used for internal branching block similarity measurement.

Index	0	1	2	...	16	17	...	362	363	364	...	n
Info.	Local variable used	Parameter variable used	Operator used			C library function used			Unique UDF* count match	Unique string/ number/functions		

UDF*: User-Defined Function

that do not fall into these categories are mostly simple variable assignments, and considering that variable names are not particularly helpful for cross-comparison, the proportion of control blocks can be regarded as sufficiently high. The slightly lower coverage in binaries is primarily due to the decompilation process, which introduces a large number of trivial variable assignments. However, these assignments do not contribute meaningful information when mapping to the source code, and thus considering them would even degrade performance. This observation highlights that control blocks provide a more reasonable and effective abstraction for source-to-binary matching.

For example, Listing 5 illustrates the process of block extraction. As in the case of a `Condition` block (*i.e.*, line #8), the block contents may consist of a single line of code, whereas, as in the case of a `Loop` block (*i.e.*, lines #10 to #12), it may span multiple lines. If a function makes a recursive call, it is defined as a `RecurFunction` (*i.e.*, line #24), while calls to other functions are defined as `CalleeFunction` (*i.e.*, lines #17 and #18). A single line of code can also belong to multiple control blocks. For example, if there is a statement "`printf("hello");`", the string literal "`hello`" belongs both to a `String` block and to the block content of a `LibcFunction`.

Function normalization. SBridge further applies function normalization, ensuring more precise function matching. To ensure the generality of SBridge, we avoid applying overly heuristic normalization rules and instead focus on clear transformations that address changes frequently introduced during compilation. SBridge first (1) removes all comments from the source functions. Next, it (2) replaces all `ASCII` and hexadecimal characters with their decimal representations, to account for the frequent character-type conversions that occur during compilation (*e.g.*, `0x5E` $\longrightarrow$ `94`). Finally, (3) every occurrence of variable names and parameter names that are not used in cross-comparison is replaced with the keywords `LVAR` and `PARAM`, respectively. For all blocks, both the key features and block content are preserved in the normalized form.

3.2 Control Block Similarity Measurement (P2)

Next, we introduce SBridge's approach for computing similarity between two blocks. Here, comparisons between block types that have little chance of mapping (*e.g.*, `Condition` *vs.* `String`) can impair the performance and accuracy. Therefore, SBridge compares blocks of the similar type: it compares C1, C2, and C3 with each other (called *internal branching* blocks), compares C5, C6, and C7 (called *external branching* blocks), and compares C4 (*string* blocks) separately.

SBridge applies fine-grained similarity analysis to *internal-branching* blocks, as they span multiple lines and embody diverse control-flow semantics with conditional variations. In contrast, *external-branching* and string blocks are typically short (*i.e.*, one or two lines), thus SBridge adopts a categorical scheme for these blocks: exact match based on key features (similarity = 1), partial match based on block contents (similarity = 0.5), and no match (similarity = 0).

Internal branching blocks: Condition, Else-Condition, and Loop. For internal branching blocks (C1 to C3), SBridge employs a *feature-based vector representation* approach that abstracts compilation-specific variations to focus on features essential for accurate matching.

Table 3 shows the overall vector structure that consists of two components: *static components* that capture predetermined features (indices 0 - 363) and *dynamic components* that contain context-specific information (indices 364 - *n*). All values in the vector are set using binary encoding (0 or

Table 4. Operator groups in condition expressions (14 groups).

Group	Operators
equOpr	equals, notEquals
comOpr	lessThan, greaterThan, lessEqualsThan, greaterEqualsThan
addOpr	addition, assignmentPlus
subOpr	subtraction, assignmentMinus
mulOpr	multiplication, assignmentMultiplication
divOpr	division, assignmentDivision
modOpr	modulo, assignmentModulo
notOpr	not, logicalNot

Group	Operators
xorOpr	xor, logicalXor
shiftOpr	shiftLeft, arithmeticShiftLeft, shiftRight, arithmeticShiftRight, assignmentArithmeticShiftLeft, assignmentArithmeticShiftRight
logAndOrOpr	logicalAnd, logicalOr
bitAndOrOpr	or, and
accOpr	indirectFieldAccess, fieldAccess, indirectIndexAccess
infOpr	infiniteLoop
assignOpr	assignment

1), because the frequency of certain elements may vary during compilation and decompilation. This approach emphasizes existence rather than frequency, while also enhancing computational efficiency.

For each internal branching block, separate vectors are generated for its key features and for its block content. The details for each element are as follows.

- **Local variable used.** The value is set to 1 if a local variable is used; otherwise, it is 0.
- **Parameter variable used.** The value is set to 1 if a parameter value is used, and 0 otherwise.
- **Operator used.** These elements capture operators. However, an operator can be replaced with another expression that carries a similar semantic meaning during compilation. Therefore, we group semantically similar operators (see Table 4), and set each vector index to 1 if an operator from the corresponding group is used, and 0 otherwise.
- **C library function used.** For the 346 predefined standard C library functions (see Section 3.1), each corresponding vector entry is set to 1 if the function is used and 0 otherwise.
- **Unique user-defined function count match.** Whether the count of unique user-defined functions matches between the compared blocks. This feature is computable only by comparing a source block with a candidate binary block. For internal-branching blocks extracted from the source function, SBridge fixes the value to 1. For internal-branching blocks in the candidate binary function, SBridge sets the value to 1 if the number of distinct user-defined functions in the binary block equals that of the source block; otherwise, 0.
- **Unique string/number/functions.** Three important but non-fixed elements (*i.e.*, strings, numeric information, and invoked function information) are additionally incorporated into the vector dynamically. For each of these elements, a new vector entry is created and its value is set to 1 if present; otherwise, it remains 0 when absent from the current vector but present in the compared one. In practice, only a small number of these elements are added, with only a marginal increase in vector size and negligible overhead during block similarity computation.

To compute the similarity between two blocks, SBridge compares the vectors extracted from their key features and block contents, respectively. Given two vectors to be compared, SBridge first performs feature space alignment: If one vector contains dynamically added entries that are absent in the other, the same entries are appended to the latter with their values set to 0.

SBridge then measures the similarity between the two vectors, denoted as $\mathbf{v}_X$ and $\mathbf{v}_Y$, using *cosine similarity* (ϕ). The cosine similarity between two vectors is calculated as follows.

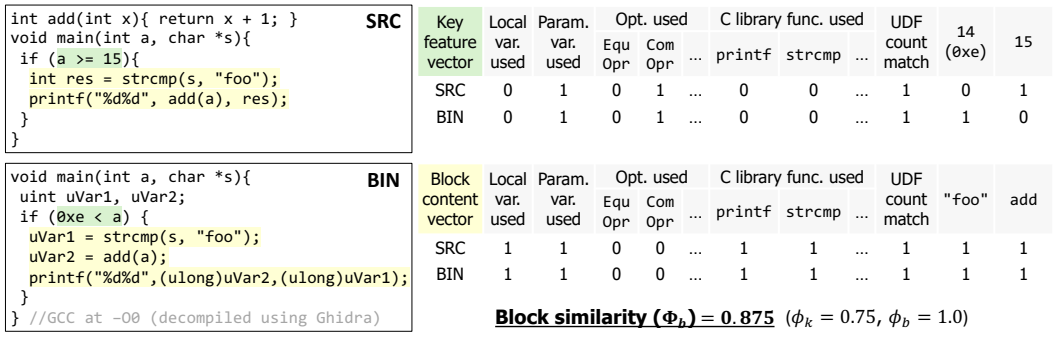


Fig. 4. Flow of internal branching block vector extraction and similarity comparison for the example code.

$$\phi(\mathbf{v}_X, \mathbf{v}_Y) = \frac{\mathbf{v}_X \cdot \mathbf{v}_Y}{||\mathbf{v}_X||\,||\mathbf{v}_Y||} = \frac{\sum_{i=1}^{m} (\mathbf{v}_X)_i \times (\mathbf{v}_Y)_i}{\sqrt{\sum_{i=1}^{m} ((\mathbf{v}_X)_i)^2} \times \sqrt{\sum_{i=1}^{m} ((\mathbf{v}_Y)_i)^2}}$$

Using this approach, SBridge computes the similarity of the key feature vectors (ϕ_k) and the similarity of the block content vectors (ϕ_b) for each pair of internal branching blocks. Let the two blocks to be compared be denoted as B_{I_1} and B_{I_2}. The final block similarity (Φ_b) is defined as the average of the key feature similarity and the block content similarity.

$$\Phi_b(B_{I_1}, B_{I_2}) = (\phi_k + \phi_b)/2$$

Although key features typically consist of a single line, block contents may span multiple lines. Averaging both measures emphasizes the importance of key features, which capture the core semantics despite their brevity. For `Else-Condition` blocks, which do not include a key feature, only the similarity derived from the block content is used.

Figure 4 illustrates SBridge's similarity computation on the example code. Although the conditional expressions differ (`a >= 15` and `0xe < a`) and most block statements are modified, SBridge's approach produced a similarity of 0.875. In contrast, cosine similarity computed on space-separated tokens and `Levenshtein` distance–based similarity yielded similarity scores of 0.3640 and 0.4863, respectively, highlighting SBridge's effectiveness in cross-domain block similarity analysis.

External branching blocks: LibcFunction, CalleeFunction, and RecurFunction. Here, SBridge first examines the key features by comparing the function name and the number of parameters. If both match, the block similarity $\Phi_b = 1$. However, when the binary is stripped, function names may be altered. Therefore, if the key features do not match, SBridge examines the parameter information (*i.e.*, block contents). Considering the order, if both the parameter values and types match, we set $\Phi_b = 1$; if only the types match, $\Phi_b = 0.5$; otherwise, $\Phi_b = 0$.

String blocks. For `string` blocks (C4), because string literals rarely change, they are directly compared. If they match, the block similarity $\Phi_b = 1$; otherwise, $\Phi_b = 0$.

3.3 Control Block-Based Function Matching (P3)

SBridge computes initial function similarity (Φ_f) by counting the number of similar blocks between the source function (f_s) and the binary function (f_b). Let θ_b be the threshold for block similarity. For function similarity computation, each source block is matched to at most one binary block. If multiple binary blocks exceed θ_b for a source block, SBridge selects the highest-similarity pair, and the matched source and binary blocks are excluded from subsequent pairing to avoid duplicate

matches.

$$\Phi_f(f_s, f_b) = \frac{|\{\, B_s \in f_s \mid \exists\, B_b \in f_b,\ \Phi_b(B_s, B_b) \geq \theta_b|}{|\{\, B_s \in f_s\}|}$$

Addressing function inlining. SBridge handles function inlining through *length-based weighting*. Although large length differences between functions usually lower similarity, SBridge avoids penalizing cases where inlining makes a binary function much larger than its source, since the source is still fully included.

To this end, SBridge employs a structured matching strategy based on the source call graph. Starting from the root, it performs top-down matching via depth-first search (DFS), aligning binary functions with their source counterparts. When a match is found, two conditions are checked: (1) whether the source function has child nodes, and (2) whether the binary function includes external calls (C5 block). If both hold, this indicates a low probability of inlining; if the source has children but the binary has no external calls, this indicates a high probability of inlining, suggesting that the child may have been merged into the parent.

To support this process, SBridge defines the following function-length weighting mechanism (w_{length}). In this equation, $\texttt{\#blocks}(f)$ represents the number of control blocks in the function f.

$$w_{length} = \frac{1.0}{1.0 + \log\left(\texttt{\#blocks}(f_b)/\texttt{\#blocks}(f_s) + 1.0\right)}$$

A logarithmic function is applied to mitigate extreme differences in the number of blocks between binary and source functions, where the "+1.0" term ensures valid computation even when $\texttt{\#blocks}(f_b)$ is zero. If only one source function exists or no call relationships are present, SBridge sets $w_{length} = 1$. When the preceding analysis indicates a low probability of inlining and the binary function has substantially more blocks, w_{length} decreases to reduce the similarity score. In contrast, when the analysis suggests inlining, SBridge fixes $w_{length} = 1$ so that length differences do not affect the computation. Section 5 provides a more detailed discussion on function inlining.

Function similarity. The final function similarity Φ is calculated as follows.

$$\Phi(f_s, f_b) = w_{length} \cdot \Phi_f$$

Based on these similarity scores, SBridge ranks candidate binary functions and identifies the binary function that is most similar to the input source function.

4 Evaluation

In this section, we evaluate SBridge based on the following four questions.

RQ1. Accuracy: How accurately does SBridge detect binary counterparts of source functions?
RQ2. Efficacy: How effectively does SBridge handle function inlining?
RQ3. Performance: How fast and scalable is SBridge in detecting similar functions?
RQ4. Application: How effective is SBridge when used for vulnerability detection?

SBridge is implemented in F# with 4.1K lines of code (LOC). It uses Ctags [6] and Joern parser [48] for function extraction and normalization. Although compatible with various binary analysis tools, SBridge adopts Ghidra [2], an open-source tool, to avoid licensing constraints. Experiments ran on Ubuntu 22.04, Intel Core Ultra 7 265K processor clocked at 3.9GHz, 32GB RAM, and 1TB SSD.

4.1 Accuracy of SBridge

Dataset. We used the binary code similarity analysis (BCSA) benchmark from BinKit [22]. We selected the top two GNU software packages in BinKit with the most binaries: GNU Coreutils (v9.0) and Inetutils (v2.4). Initially, 122 target packages were collected from this benchmark.

To evaluate SBridge across diverse environments, we compiled each binary under 32 configurations, derived from four architectures (`ARM32`, `ARM64`, `x86`, `x64`), two compilers (`GCC` 9.4.0 and `Clang` 13.0), two optimization levels (`-O0` and `-O2`), and two symbol management options (`strip` and `no_strip`). This produced 3,904 binaries containing 624,192 functions for accuracy evaluation.

Methodology. We evaluate SBridge by using the source codes of 122 collected packages, comprising 1,618 source functions, as input. The goal is to assess how accurately SBridge identifies source functions (f_s) that have been compiled into binaries (f_b) under various environments. If SBridge identifies f_b as similar to f_s, it is considered a true positive (TP). For ground truth, we extracted function mappings from `no_strip` and `O0` binaries based on function names. When multiple functions shared the same name, we manually examined the source and binary functions. In this process, we assume functions in `O0` binaries are not inlined. Although this assumption may introduce minor discrepancies, it ensures a consistent basis for comparison across all tools. In our experiments, we set the block similarity threshold θ_b to 0.7 (see Section 3.3), and analyze its sensitivity at the end of the accuracy evaluation.

Comparison targets. For accuracy evaluation, we compared SBridge with two state-of-the-art approaches representing the source-to-source and source-to-binary domains, both of which are publicly available and thus suitable for direct comparison: MRT-OAST [53] (source-to-source) and BinaryAI [20] (source-to-binary). MRT-OAST is a deep learning–based code clone detection technique that leverages optimized abstract syntax trees (OAST) and a Siamese network. We employed MRT-OAST to identify code clones between input source code and decompiled binary code, and then measured its accuracy. BinaryAI [20] detects reused components by mapping binary functions to source functions. For our evaluation, we adapt BinaryAI for function-level evaluation by reversing its process. Specifically, when it identifies a source function (f_s) similar to a binary function (f_b), we invert the mapping ($f_s \rightarrow f_b$) and use this to compare against SBridge. The potential limitations of this adaptation are discussed in Section 5.

Both approaches can be applied to detecting modified code clones, thereby addressing our first challenge (*i.e.*, loss of source code context) to some extent. However, MRT-OAST falls short in tackling the second and third challenges (*i.e.*, architecture and compiler variability and addressing function inlining), while BinaryAI does not sufficiently consider inlining and thus struggles with the third challenge. For these reasons, we considered them appropriate baselines for comparison.

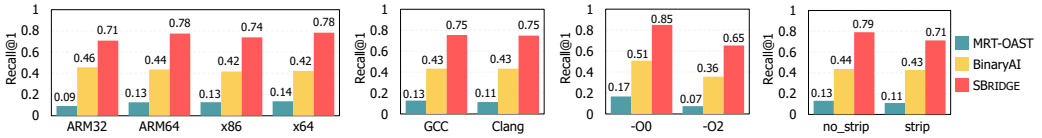
Evaluation metric. We use `Recall@1`, `Recall@5`, and Mean Reciprocal Rank (`MRR`) for accuracy evaluation (TP = true positive, FN = false negative).

$$\texttt{Recall@1} = \frac{\texttt{\#TP@1}}{\texttt{\#TP@1} + \texttt{\#FN@1}}, \quad \texttt{Recall@5} = \frac{\texttt{\#TP@5}}{\texttt{\#TP@5} + \texttt{\#FN@5}}, \quad \texttt{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

`TP@1` denotes that the correct binary function (*i.e.*, compiled from the source function) is ranked first for the given source function, while `TP@5` means it appears within the top five. Otherwise, the case is regarded as `FN@1` or `FN@5`, respectively. Because each source function is evaluated against only the top-k candidates, an incorrect match (false positive; FP) necessarily means the correct one is missed (FN). Hence, FP and FN coincide, and `Recall@k` is used as the evaluation metric [20]. `MRR`

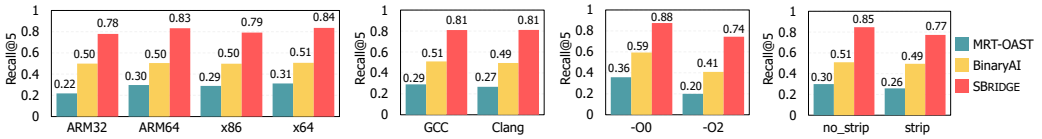
Table 5. The accuracy measurement results by architecture, compiler, optimization, and symbol management for the three tools. We consider only input source functions for which the ground truth could be measured. The bold results indicate the highest recall for each configuration (R@1: Recall@1, R@5: Recall@5).

	MRT-OAST			BinaryAI			SBridge		
	R@1	R@5	MRR	R@1	R@5	MRR	R@1	R@5	MRR
By architecture									
ARM32	0.0918	0.2188	0.1651	0.4567	0.4994	0.4763	**0.7082**	**0.7787**	**0.7718**
ARM64	0.1270	0.2969	0.2149	0.4364	0.5045	0.4679	**0.7751**	**0.8325**	**0.8275**
x86	0.1276	0.2885	0.2127	0.4175	0.4979	0.4554	**0.7394**	**0.7921**	**0.7869**
x64	0.1360	0.3115	0.2280	0.4220	0.5062	0.4618	**0.7827**	**0.8358**	**0.8299**
By compilers									
GCC	0.1276	0.2906	0.2148	0.4329	0.5099	0.4688	**0.7537**	**0.8097**	**0.8051**
Clang	0.1137	0.2672	0.1955	0.4333	0.4941	0.4619	**0.7490**	**0.8099**	**0.8029**
By optimizations									
O0	0.1672	0.3583	0.2653	0.5087	0.5933	0.5482	**0.8496**	**0.8754**	**0.8802**
O2	0.0741	0.1995	0.1450	0.3560	0.4088	0.3808	**0.6531**	**0.7441**	**0.7279**
By symbol managements									
no_strip	0.1310	0.2992	0.2189	0.4378	0.5092	0.4713	**0.7909**	**0.8470**	**0.8375**
strip	0.1102	0.2586	0.1914	0.4284	0.4948	0.4594	**0.7119**	**0.7725**	**0.7706**
Total result (average)									
Total	0.1206	0.2789	0.2052	0.4331	0.5020	0.4653	**0.7513**	**0.8098**	**0.8040**



(a) By architecture. (b) By compiler. (c) By optimization. (d) By symbol management.

Fig. 5. Recall@1 measurement results by architecture, compiler, optimization, and symbol management.



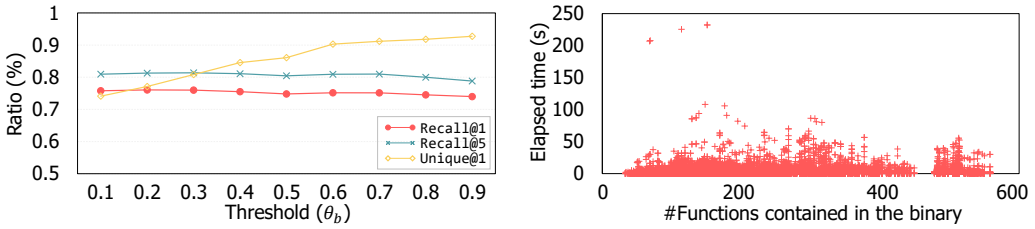
(a) By architecture. (b) By compiler. (c) By optimization. (d) By symbol management.

Fig. 6. Recall@5 measurement results by architecture, compiler, optimization, and symbol management.

is used to evaluate how highly the correct results are ranked for a given query. In our setting, the query is an input source function, and the correct result is its corresponding binary function.

Result overview. Experimental results show that SBridge outperforms both the MRT-OAST and BinaryAI across most configurations. In particular, SBridge achieved 6.23 times higher Recall@1 compared to the MRT-OAST and 1.73 times higher Recall@1 compared to BinaryAI. Table 5, Figure 5, and Figure 6 illustrate the detection results of MRT-OAST, BinaryAI, and SBridge.

Result analysis: existing approaches. First, we observed that function inlining has the most significant impact on the accuracy of existing approaches. This is evident from the optimization



(a) Experimental results for θ_b sensitivity. `Unique@1` represents the proportion of cases where the top-1 result is uniquely ranked without ties.

(b) Matching time per source function according to the number of binary function candidates. 96.8% of the cases were matched in under 10 s on average.

Fig. 7. Threshold experiment and performance evaluation results.

experiments: the `Recall@1` of MRT-OAST decreased from 0.1672 (at `-O0`) to 0.0741 (at `-O2`), while that of BinaryAI dropped from 0.5087 to 0.3560 (see Figure 6c).

In particular, MRT-OAST relies on syntactic information, and thus its identification accuracy decreases when binaries are stripped. Specifically, its `Recall@1` dropped from 0.1310 in the `no_strip` setting to 0.1102 in the `strip` setting.

BinaryAI achieved higher `Recall@1` and `Recall@5` compared to MRT-OAST (see Table 5). Notably, the `Recall@1` and `Recall@5` of BinaryAI showed little difference. This is because the tool was not originally designed to identify similar binary functions for a given source input, and even after our adaptation, it often produced fewer than five candidate binaries per source function. Overall, the accuracy of BinaryAI remained consistent, except when the `-O2` optimization was applied.

Result analysis: SBridge. SBridge outperformed existing approaches by achieving 0.7513 and 0.8098 `Recall@1` and `Recall@5`, respectively.

Despite its effectiveness, SBridge occasionally produced false results due to inaccurate decompilation or extreme code transformations. Although it substantially mitigates the effects of inlining, its accuracy under the `-O2` optimization slightly decreased because of extreme code logic modifications. However, the `Recall@5` drop rate of SBridge from `-O0` to `-O2` was 14.99%, which is considerably smaller than that of MRT-OAST (44.31%) and BinaryAI (31.09%). In addition, because SBridge incorporates syntactic features in block comparison, its accuracy was slightly lower on stripped binaries than on `no_strip` binaries. Moreover, exceptionally long decompiled functions sometimes caused incorrect mappings even though length-based filtering was applied in P3. Nevertheless, SBridge achieved the highest accuracy across all configurations, confirming that its design is well-suited for measuring similarity between source and binary functions.

Finding 1. SBridge outperformed the MRT-OAST and BinaryAI by achieving higher `Recall@1` and `Recall@5` in all configurations. The block-based function similarity matching of SBridge achieved substantially higher detection accuracy than existing approaches, even under `strip` and `-O2` optimization settings.

Threshold sensitivity. To measure threshold sensitivity, we varied θ_b from 0.1 to 0.9 in increments of 0.1 and evaluated SBridge's recalls. Moreover, as θ decreases, the matching becomes looser, which may cause two unrelated blocks to be determined as similar. To analyze this effect, we introduce a metric called `Unique@1`, which represents the proportion of cases where exactly one binary function is identified as the most similar to a given source function. Figure 7a presents the experimental results. Notably, SBridge's accuracy did not vary significantly with different values of θ_b. However, when the threshold was set too high (*e.g.*, 0.9), similar blocks could not be detected,

Table 6. Accuracy of SBridge in inlined function detection.

Testset	Compiler	Optimization	Inline †(%)	Symbol.	Recall@1	Recall@5
Coreutils	Clang	O2	54.12%	no_strip	0.5981	0.7315
	Clang	O2	54.12%	strip	0.5624	0.6712
	GCC	O2	45.65%	no_strip	0.5930	0.7343
	GCC	O2	45.65%	strip	0.5284	0.6595
Inetutils	Clang	O2	11.33%	no_strip	0.7297	0.7613
	Clang	O2	11.33%	strip	0.6441	0.7162
	GCC	O2	9.29%	no_strip	0.5879	0.6593
	GCC	O2	9.29%	strip	0.6319	0.6593

†: The proportion of inlined functions among all binary functions.

leading to a slight decrease in recall, whereas when it was set too low (*e.g.*, below 0.6), unrelated blocks were identified as similar, resulting in a decrease in Unique@1. Therefore, we set θ_b to 0.7 to achieve high recall while effectively distinguishing similar functions from non-similar ones.

4.2 Efficacy of SBridge

Next, we evaluate how effectively SBridge handles function inlining. Within the experimental results of Section 4.1, we considered only inlined functions and examined how well SBridge identified them. Table 6 presents the measurement results. For Coreutils and Inetutils, we observed that when compiled with O2 optimization, approximately 50% and 10% of the functions, respectively, were inlined. This is because function inlining decisions depend on various factors (*e.g.*, function size and call frequency) and occur in an unpredictable manner.

Nevertheless, SBridge successfully identified the corresponding binary functions for input source functions in binaries where both function inlining and stripping were applied, achieving a Recall@1 of at least 0.5284 (up to 0.6441) and a Recall@5 of at least 0.6593 (up to 0.7613). When the binary is not stripped, the maximum Recall@1 and Recall@5 increase to 0.7297 and 0.7613. Existing approaches have not addressed function inlining in function mapping. For example, BinaryAI achieved a Recall@1 of less than 0.01 (*i.e.*, 1%) when applied solely to inlined functions. Therefore, this result strongly demonstrates SBridge's effectiveness in identifying inlined functions.

Finding 2. SBridge, which utilizes block-based function matching and a length-based weighting mechanism, can identify inlined functions with achieving up to 0.7297 Recall@1 and 0.7613 Recall@5, whereas existing approaches struggle to detect them (*e.g.*, less than 0.01 Recall@1).

4.3 Performance of SBridge

To evaluate the performance and scalability of SBridge, we measured the time required to identify binary functions similar to the corresponding input source code using 3,904 binaries from the accuracy measurement experiment (see Section 4.1). We excluded preprocessing steps that use external libraries (*e.g.*, the time taken by the Joern parser) and considered only the time required to identify similar functions, as preprocessing time depends on external tools and their implementations. We can measure the execution time either based on the binary size or on the number of functions contained in the binary. However, the former is highly variable due to factors such as optimization or debugging information (*i.e.*, external influences). Therefore, we measured the execution time of SBridge based on the latter (*i.e.*, the number of functions), as it directly determines the number of candidate comparisons and thus dominates the overall matching cost.

Figure 7b illustrates the measurement result. When comparing a single input source function with a single binary (with multiple functions), SBridge took an average of 1.9 s (median 0.692 s). For 96.8% of the input functions, SBridge took less than 10 s, which demonstrates that SBridge is sufficiently fast for practical use. Next, even as the number of functions in the binary increased, the mapping time did not increase significantly. Therefore, unless a block contains an extreme amount of block content (*e.g.*, cryptography functions), the performance of SBridge remains fast. This demonstrates that SBridge can operate efficiently and scale well even for large binaries.

Finding 3. With an average identification time of 1.9 s to detect similar functions in the target binary, SBridge demonstrates performance that is sufficient for practical use.

4.4 Application of SBridge

In this section, we applied SBridge to 1-day vulnerability detection and evaluated its effectiveness.

Methodology. We compared SBridge with React [55], a state-of-the-art approach for checking whether a vulnerability patch has been applied in binaries. React identifies the target binary function based on the name of the vulnerable function, and then verifies the patch application using intermediate representations (IR). We apply SBridge to vulnerability detection by first locating the binary function most similar to the vulnerable function. For a fair comparison, if the function name is identifiable, we utilize this information. SBridge then focuses on the blocks in the vulnerable (*resp.* patched) function that contain deleted (*resp.* added) lines from the patch [38, 39]. After mapping these blocks to all blocks of the target binary function, SBridge computes the highest similarity scores for the vulnerable and patched blocks, denoted as α and β, respectively. If $\alpha \geq \beta$, SBridge determines that the vulnerability is present in the binary.

To evaluate the practical applicability of our approach to vulnerability detection, we compiled both vulnerable and patched binaries for each selected CVE in eight configurations (2 compilers $\times$ 2 optimizations $\times$ 2 symbol management options). React was evaluated on four OSS projects: `LibXML2`, `tcpdump`, `OpenSSL`, and `FFmpeg`. We utilized all CVEs for `LibXML2` and `tcpdump` from the React's dataset, as their binaries are relatively small and contain a manageable number of functions. In contrast, `OpenSSL` and `FFmpeg` presented a significant challenge due to their large scale and high variance between patched versions. Therefore, we selected only CVEs whose patched binaries belonged to the latest major version within React's dataset. This process resulted in 416 binaries from 26 CVEs (208 vulnerable and 208 patched). We determined that this dataset is sufficient for our purposes, because the primary goal of this experiment is to show that SBridge, our tool for detecting similar functions, can effectively be extended to vulnerability detection.

Detection results are classified as follows: correctly identifying a vulnerable function in vulnerable binaries (TP), failing to detect it (FN), incorrectly flagging a patched binary as vulnerable (FP), and correctly identifying a patched binary as safe (TN).

We evaluated SBridge and React based on the following three metrics: `precision` (P), `recall` (R), and `F1 score` (F1).

$$P = \frac{\#\mathsf{TP}}{\#\mathsf{TP} + \#\mathsf{FP}}, \quad R = \frac{\#\mathsf{TP}}{\#\mathsf{TP} + \#\mathsf{FN}}, \quad \mathsf{F1} = \frac{2 * \mathsf{P} * \mathsf{R}}{\mathsf{P} + \mathsf{R}}$$

Result analysis. Despite applying SBridge to vulnerability detection in an intuitive manner, SBridge demonstrated slightly higher accuracy (*i.e.*, `F1 score`) than React. Notably, SBridge identified more TPs and fewer FNs than React. The primary cause of the high FN rate in React was its reliance on function names, which made it nearly infeasible to test the presence of patches in stripped binaries. Furthermore, when function inlining occurred under O2 optimization, React's

Table 7. Vulnerability detection results (SBridge vs. React).

OSS	#CVEs	React							SBridge						
		TP	FP	TN	FN	Precision	Recall	F1 score	TP	FP	TN	FN	Precision	Recall	F1 score
tcpdump	10	34	20	14	92	0.6296	0.2698	0.3778	55	23	39	43	0.7051	0.5612	0.6250
OpenSSL	8	32	7	25	64	0.8205	0.3333	0.4741	30	22	11	65	0.5769	0.3158	0.4082
LibXML2	2	8	4	4	16	0.6667	0.3333	0.4444	13	6	6	7	0.6842	0.6500	0.6667
FFmpeg	6	20	4	18	54	0.8333	0.2703	0.4082	18	28	6	44	0.3913	0.2903	0.3333
Total	26	94	35	61	226	0.7287	0.2938	0.4187	116	79	62	159	0.5949	0.4218	0.4936

accuracy dropped significantly. In contrast, SBridge could trace vulnerable functions even in O2-optimized or stripped binaries, yielding fewer FNs. However, because code similarity–based vulnerability detection is less effective when only small portions of the vulnerable code are modified in the patch [23, 39], it produced many FPs. Identifying 1-day vulnerabilities in stripped or O2-optimized binaries based on vulnerable source functions remains a challenging open problem. Our attempt to apply SBridge to this task achieved reasonable accuracy and demonstrates the potential of this approach as a new direction for vulnerability detection.

Finding 4. SBridge's technique for detecting similar functions across domains can also be effectively utilized for identifying propagated vulnerabilities.

5 Discussion

Addressing function inlining. SBridge addresses function inlining through two main ideas: (1) leveraging source code as a reference and (2) applying a length-based weighting mechanism. When computing function similarity (see Section 3.3), the reference point is the total number of blocks in the source function. Thus, as long as a sufficient portion of the source blocks is preserved in the binary function, it can still be identified as a similar match. Moreover, by incorporating a length-based weighting mechanism, SBridge remains robust even when inlining increases the size of the binary function compared to the source function. Together, these strategies enable SBridge to effectively handle common inlining scenarios.

String block ablation. Although string literals are among the most robust features across compilation, they do not directly belong to control flow. To evaluate string block-specific impact, we conducted an ablation study that excludes string blocks. The results showed only a minimal recall drop of no more than 3%, indicating that string blocks are not critical to matching accuracy, but instead provide a complementary benefit.

Limitations and future work. First, SBridge depends on the quality of decompilation. If the decompiler fails to accurately reconstruct source-level semantics, especially under aggressive compiler optimizations, the resulting mismatches can lead to false mappings. Next, although SBridge mitigates some effects of compiler optimizations and symbol removal, challenges remain in handling function inlining and stripped binaries. These factors still cause slight accuracy degradation. We plan to incorporate semantic features beyond syntactic block comparison (*e.g.*, data-flow) to improve robustness in such environments. Lastly, we only considered `O0` and `O2` optimizations. We assumed that `O1` would be covered by `O2` and chose `O2` over `O3` because it is more widely used and better suited to illustrating cases where function inlining and syntax transformations significantly differ. If this becomes a concern, we will extend our evaluation to include `O1` and `O3` optimizations.

Threats to validity. To demonstrate the generality of SBridge, we selected the two most binary-rich datasets from BinKit. However, these may not fully represent the entire software ecosystem. Next, due to the absence of ground truth, we created our own ground truth under the assumption that `O0` binaries are not inlined (see Section 4.1). This assumption may have a minor impact on the accuracy measurement. In addition, we adapted MRT-OAST and BinaryAI for our purposes to conduct comparative experiments. While we aimed to ensure fairness in our experiments, there may be cases where existing approaches' accuracy might not be correctly evaluated. Our goal is not to undermine them but to demonstrate that SBridge effectively identifies binary functions similar to a given source code. Next, although SBridge detected similar binary functions for the input function in an average of 1.9 s (see Section 4.3), this time may vary depending on the environment. Finally, although SBridge can utilize any decompilation tool, we used Ghidra in our experiments. If a different tool is used, the accuracy may vary slightly.

6 Related Work

Source-to-binary matching. Existing source-to-binary approaches (*e.g.*, [4, 9–11, 20]) have primarily focused on identifying reused OSS components in binaries by leveraging compilation-resilient features (*e.g.*, string literals). ISRD [46], LibDB [34], LibAM [24], and ModX [49] used function-level similarity features to identify OSS components. LibvDiff [9] focuses on OSS version identification in binaries by combining source-level differences with function-level binary similarity. REACT [55] verified patch presence by employing symbolic execution on IR code from both source and binary. BinaryAI [20] attempted to identify reused OSS components by measuring code similarity between the embeddings of decompiled binary code and source functions. CodeCMR [52] addresses the representation disparity by using node embeddings with DPCNN and GNN. CrossCode2Vec [50] addresses the compilation gap through unified embeddings across source code and its corresponding compiled binary. However, existing techniques either take a coarse-grained approach or rely on limited features, making them unsuitable for analyzing similarities between source and binary functions. Even the recent AI-based method fails to adequately handle function inlining, rendering it ineffective for our target problem.

Binary-to-binary matching. Existing binary-to-binary matching approaches aimed to assess structural or syntactic similarity by leveraging text hashing of assembly code [7, 43], matching control flow graphs [5], or performing symbolic execution [12, 29, 54]. However, these approaches face scalability limitations due to obfuscation, compiler optimizations, and path explosion issues. Alternatively, deep learning models, including Transformers and graph neural networks (GNNs), have been introduced, improving the performance of binary code similarity measurement [13, 14, 25, 28, 35, 44, 56]. Recent approaches, such as Binshot [3] and OrderMatters [51], employ pre-trained BERT models. BinXray [47] identifies patches by extracting signatures through comparison of vulnerable and patched functions. However, these approaches are also difficult to apply effectively to our target problem: they (1) are not applicable to bridge the representation gap between binaries and source code and (2) hardly consider function inlining.

7 Conclusion

Identifying reused source code in binaries is a pressing issue, but the significant gap between source and binary representations makes it highly challenging. In response, we present SBridge, an approach that effectively identifies binary functions similar to given source functions. By leveraging control block-based function matching, SBridge outperforms existing approaches in mapping source and binary functions. This enables SBridge to remain effective even when direct source-to-binary correspondence is obscured by compiler transformations, such as optimization

and function inlining. With SBridge, reused source code in binaries can be identified with high accuracy and scalability, achieving 75.13% Recall@1 and 80.98% Recall@5, with an average matching time of 1.9 seconds per source function. Furthermore, as our experiments demonstrate, SBridge can also be applied to detect propagated vulnerabilities, contributing to a safer software ecosystem.

Data Availability

SBridge is available at https://github.com/heedongy/SBridge_Artifact.

Acknowledgments

This work was supported by the Institute of Information& Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains), ICT Creative Consilience Program (IITP-2026-RS-2020-II201819), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00517788, Research on Intelligent SBOM Generation and Automated Vulnerability Analysis through Multi-level Code Analysis) and the Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism (International Collaborative Research and Global Talent Development for the Development of Copyright Management and Protection Technologies for Generative AI, RS-2024-00345025).

References

[1] Vector 35. 2024. Binary Ninja. https://binary.ninja/.
[2] National Security Agency. 2024. Ghidra. https://ghidra-sre.org.
[3] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 361–374. https://doi.org/10.1145/3564625.3567975
[4] Gu Ban, Lili Xu, Yang Xiao, Xinhua Li, Zimu Yuan, and Wei Huo. 2021. B2SMatcher: fine-Grained version identification of open-Source software in binary files. *Cybersecurity* 4 (2021), 1–21. https://doi.org/10.1186/s42400-021-00085-7
[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. https://doi.org/10.1145/2430553.2430557
[6] Ctags. 2024. Universal Ctags. https://github.com/universal-ctags/ctags.
[7] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 79–94. https://doi.org/10.1145/3140587.3062387
[8] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. 131–141.
[9] Chaopeng Dong, Siyuan Li, Shougou Yang, Yang Xiao, Yongpan Wang, Hong Li, Zhi Li, and Limin Sun. 2024. LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. 791–802. https://doi.org/10.1145/3597503.3623336
[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2169–2185. https://doi.org/10.1145/3133956.3134048
[11] Muyue Feng, Zimu Yuan, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xue, and Wei Huo. 2020. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1038–1049. https://doi.org/10.1109/ASE.2019.00100
[12] 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. https://doi.org/10.1007/978-3-540-88625-9_16

[13] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not natural language: unlock the power of semantics-oriented graph representation for binary code similarity detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 99, 18 pages.

[14] Xu He, Shu Wang, Pengbin Feng, Xinda Wang, Shiyu Sun, Qi Li, and Kun Sun. 2024. BinGo: Identifying Security Patches in Binary Code with Graph Representation Learning. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1186–1199. https://doi.org/10.1145/3634737.3637666

[15] Hex-Rays. 2024. IDA Pro. https://hex-rays.com/ida-pro/.

[16] IBM. 2025. Standard C Library Functions Table, By Name. https://www.ibm.com/docs/en/i/7.6.0?topic=extensions-standard-c-library-functions-table-by-name.

[17] 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. https://doi.org/10.1145/3561385

[18] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. 2024. Cross-Inlining Binary Function Similarity Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 223, 13 pages. https://doi.org/10.1145/3597503.3639080

[19] 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* (Copenhagen, Denmark) *(LCTES 2024)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/3652032.3657572

[20] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 224, 13 pages. https://doi.org/10.1145/3597503.3639100

[21] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-party library dependency for large-scale sca in the c/c++ ecosystem: How far are we?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1383–1395. https://doi.org/10.1145/3597926.3598143

[22] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2023. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1661–1682. https://doi.org/10.1109/TSE.2022.3187689

[23] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[24] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, and Limin Sun. 2023. LibAM: An Area Matching Framework for Detecting Third-Party Libraries in Binaries. *ACM Trans. Softw. Eng. Methodol.* (sep 2023). https://doi.org/10.1145/3625294

[25] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 667–678. https://doi.org/10.1145/3238147.3238199

[26] LLVM Project. 2025. *LLVM Project Doxygen Documentation*. LLVM Foundation. https://llvm.org/doxygen/

[27] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. 2025. *The GNU C Library Reference Manual, for version 2.42*. https://sourceware.org/glibc/manual/2.42/pdf/libc.pdf

[28] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 309–329. https://doi.org/10.1007/978-3-030-22038-9_15

[29] 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)*. Vancouver, BC, 253–270.

[30] Yoonjong Na, Seunghoon Woo, Joomyeong Lee, and Heejo Lee. 2024. CNEPS: A Precise Approach for Examining Dependencies Among Third-Party C/C++ Open-Source Components. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. 2918–2929. https://doi.org/10.1145/3597503.3639209

[31] 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 (ICSE)*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[32] Synopsys 2025. *2025 Open Source Security and Risk Analysis Report*. Synopsys.

[33] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 104–115. https://doi.org/10.1109/SANER48275.2020.9054845

[34] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: an effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/3524842.3528442

[35] 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. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13. https://doi.org/10.1145/3533767.3534367

[36] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: A Token Based Large-Gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 1066–1077. https://doi.org/10.1145/3180155.3180179

[37] Seunghoon Woo, Eunjin Choi, and Heejo Lee. 2025. A large-scale analysis of the effectiveness of publicly reported security patches. *Computers & Security* 148 (2025), 104181. https://doi.org/10.1016/j.cose.2024.104181

[38] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. 6541–6556.

[39] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium (Security)*. 3037–3053.

[40] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (Security)*. 3041–3058.

[41] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 860–872. https://doi.org/10.1109/ICSE43902.2021.00083

[42] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of the 29th USENIX Security Symposium (Security)*. 1165–1182.

[43] Yang Xiao, Zhengzi Xu, Weiwei Zhang, Chendong Yu, Longquan Liu, Wei Zou, Zimu Yuan, Yang Liu, Aihua Piao, and Wei Huo. 2021. VIVA: Binary Level Vulnerability Identification via Partial Signature. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 213–224. https://doi.org/10.1109/SANER50967.2021.00028

[44] 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. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1106–1118. https://doi.org/10.1145/3597926.3598121

[45] Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE)*. 401–412. https://doi.org/10.1145/3611643.3616301

[46] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled Software Reuse Detection Based on a Multi-Level Birthmark Model. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE43902.2021.00084

[47] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch Based Vulnerability Matching for Binary Programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 376–387. https://doi.org/10.1145/3395363.3397361

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*. IEEE, 590–604. https://doi.org/10.1109/SP.2014.44

[49] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405. https://doi.org/10.1145/3510003.3510627

[50] Gaoqing Yu, Jing An, Jiuyang Lyu, Wei Huang, Wenqing Fan, Yixuan Cheng, and Aina Sui. 2025. CrossCode2Vec: A unified representation across source and binary functions for code similarity detection. *Neurocomputing* 620 (2025), 129238. https://doi.org/10.1016/j.neucom.2024.129238

[51] 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. https://doi.org/10.1609/aaai.v34i01.5466

[52] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: cross-modal retrieval for function-level binary source code matching. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 326, 12 pages.

[53] Yu, Tianchen and Yuan, Li and Lin, Liannan and He, Hongkui. 2025. A Multiple Representation Transformer with Optimized Abstract Syntax Tree for Efficient Code Clone Detection. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 587–587. https://doi.org/10.1109/ICSE55347.2025.00050

[54] Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. 2024. Ps3: Precise patch presence test based on semantic symbolic signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3597503.3639134

[55] Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. 2024. REACT: IR-Level Patch Presence Test for Binary. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 381–392. https://doi.org/10.1145/3691620.3695012

[56] Wenyu Zhu, Hao Wang, Yuchen Zhou, Jiaming Wang, Zihan Sha, Zeyu Gao, and Chao Zhang. 2023. kTrans: Knowledge-Aware Transformer for Binary Code Embedding. *arXiv preprint arXiv:2308.12659* (2023).

Received 2025-09-12; accepted 2025-12-22