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

Heedong Yang, Jeongwoo Lee, Hajin Yun, and Seunghoon Woo

Korea University

FSE'26

KOREA UNIVERSITY

SSP LAB
Software Security and Privacy Laboratory

Background : Reused OSS in Binaries

- Modern software heavily reuses open-source software (OSS)
- COTS binaries often include OSS components
- Reused code in binaries impacts:
  - License compliance Problem
  - Propagated 1-day vulnerability

→ Key need: Identify which similar codes are reused inside binaries

Why match Source code to Binaries?

Binary-to-Binary

- Pros
  - Most BCSD approaches are binary-to-binary
  - Preserves binary-level semantics
- Cons
  - High cost to maintain many compilation variants

Source-to-Binary

- Pros
  - Source code is easier to collect
  - Source file information can use for analyze directly
- Cons
  - Large source–binary semantic gap (names/types lost after compilation)
  - Function inlining can break 1:1 matching

Why match Source code to Binaries?

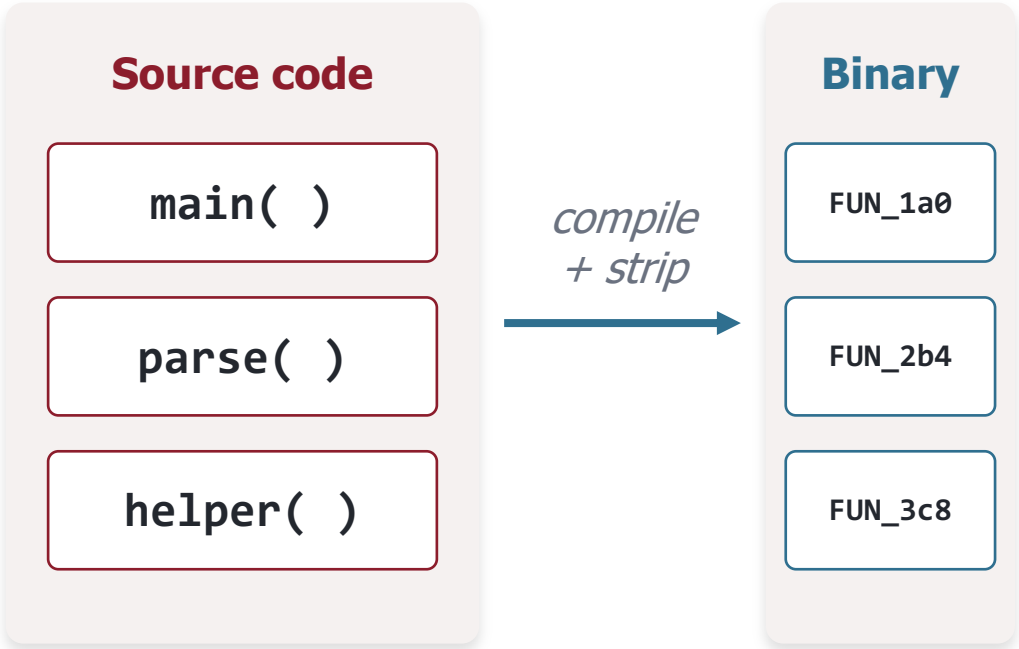
Binary-to-Binary

- Pros
  - Most BCSD approaches are binary-to-binary
  - Preserves binary-level semantics
- Cons
  - **High cost** to maintain many compilation variants

Source-to-Binary ✓

- Pros
  - Source code is **easier to collect**
  - Source file information can use for **analyze directly**
- Cons
  - Large source–binary semantic gap (names/types lost after compilation)
  - Function inlining can break 1:1 matching

Why match Source code to Binaries?

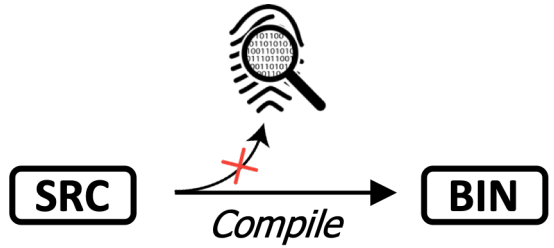


which came from which?

GOAL : Given a source function, find similar function(s) in a target binary

Challenge 1. Loss of source code context

- Compilation strips variable names, types, and structure
- Most COTS software released as stripped binaries → more difficult



Source

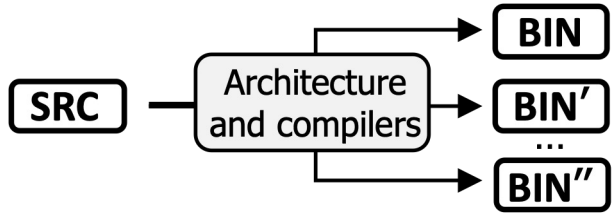
```
int check(char c) {
  if (c >= '/')
    return 1;   // pass
}
```

Binary (decompiled)

```
undefined4 FUN_4011a0(char p1) {
  if (p1 < 0x2f)
    return 0;
  return 1;
}
```

Challenge 2. Architecture & compiler variability

- Architectures, compilers, and optimization options reshape binaries
- The same source code may map to different libc calls



Same source

```
printf(
  "done\n"
);
```

Binary(Clang/O0)

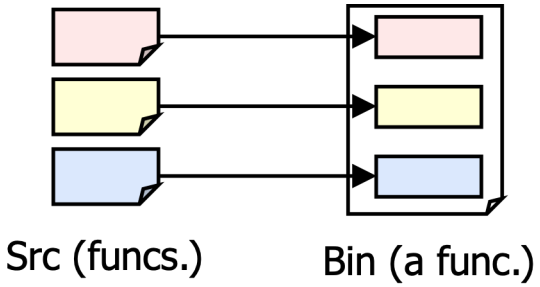
```
printf(
  "done\n"
);
```

Binary (GCC/O2)

```
puts(
  "done"
);
```

Challenge 3. Function inlining

- Compilers embed callee code at call sites for performance
- Multiple source functions merge into one binary function (N-to-1)



Source (GNU 'csplit')

```
main()
    CALL parse_patterns()
        CALL extract_regexp()
```

Binary (O2)

```
main()
    Inlined ~~parse_patterns()~~
        Inlined ~~extract_regexp()~~
```

Our Approach : SBridge

- To overcome these three challenges, we introduce **SBridge** (**S**ource-to-**B**inary br**idge**)

SBridge

Cross-Domain Control Block matching

Challenge 1 Loss of source code context

Challenge 2 Architecture & compiler variability

Challenge 3 Function inlining

Key Idea: Cross-Domain Control Block

*** Control Block ?**

Fine-grained code units that captures a distinct structural or functional component.

C1 Condition

Conditional expressions
(e.g., if and switch)

C2 Else-Condition

Alternative branches
(else conditions)

C3 Loop

Loop structures
(e.g., for and while)

C4 String

String literals
(compile-robust)

C5 LibcFunction

346 standard C-library calls
(e.g., *printf*, *scanf*)

C6 CalleeFunction

User-defined function
calls

C7 RecurFunction

Recursive calls
(inlining-immune)

- **Internal-branching block:** C1–C3, **External-branching block :** C5–C7 , **String:** C4

Key Idea: Cross-Domain Control Block

*** Control Block ?**

Fine-grained code units that captures a distinct structural or functional component.

Source function

```
void run(int n) {
  char *msg = "ok";     [C4] String
  if (n <= 0)           [C1] Condition
    return;
  while (n > 0) {       [C3] Loop
    printf("msg:");     [C5] Call (libc)
    n--;
  }
}
```

C1 Condition

KF: PVAR <= 0

BC: return

C3 Loop

KF: PVAR > 0

BC: printf("msg:"), PVAR = PVAR - 1

C4 String

KF: ok

BC: None

C5 LibcFunction

KF: printf, 1

BC: (LITERAL, "msg")

- **KF (Key Feature):** the primary information of a block
- **BC (Block Content):** supporting code context used for matching

Our Approach : SBridge

- **SBridge use a three-phase approach to address these challenges**

SBridge

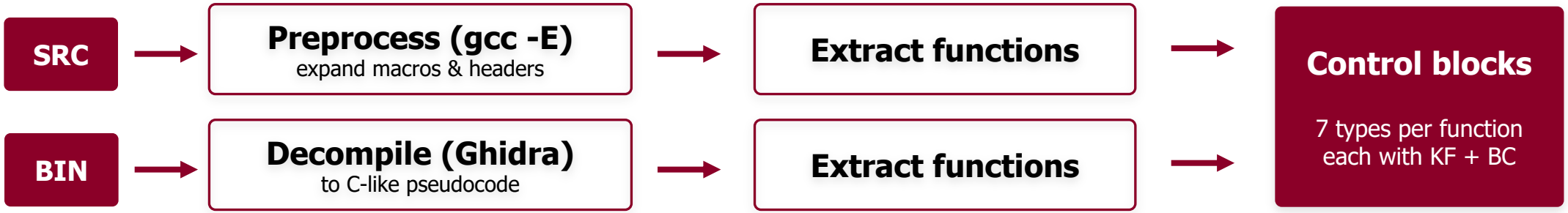
Cross-Domain Control Block matching

P1	**Control block extraction**	→	**Challenge 1** Loss of source code context
P2	**Block similarity measurement**	→	**Challenge 2** Architecture & compiler variability
P3	**Block-based function matching**	→	**Challenge 3** Function inlining

P1. Control block extraction

Solving Challenge 1 : Loss of source code context

- Extract Compile-robust blocks



- Normalization
  - variable/parameter names → LVAR / PVAR
  - ASCII & Hex value → decimal

P2. Control Block similarity

Solving Challenge 2 : Architecture & compiler variability

- Measures the similarity of *internal-branching blocks using feature-based vectors

  * Condition, Else-condition, Loop

Source code

```
int add(int x){ return x + 1; }
void main(int a, char *s){
 if (a >= 15){
  int res = strcmp(s, "foo");
  printf("%d%d", add(a), res);
 }
}
```

Key feature vector	Local var. used	Param. var. used	Opt. used			C library func. used			UDF count mtach	14 (0xe)	15
			Equ Opr	Com Opr	...	printf	strcmp	...			
SRC	0	1	0	1	...	0	0	...	1	0	1
BIN	0	1	0	1	...	0	0	...	1	1	0

Binary (GCC -O0, decompiled by Ghidra)

```
void main(int a, char *s){
 uint uVar1, uVar2;
 if (0xe < a) {
  uVar1 = strcmp(s, "foo");
  uVar2 = add(a);
  printf("%d%d",(ulong)uVar2,(ulong)uVar1);
 }
```

Block content vector	Local var. used	Param. var. used	Opt. used			C library func. used			UDF count match	"foo"	add
			Equ Opr	Com Opr	...	printf	strcmp	...			
SRC	1	1	0	0	...	1	1	...	1	1	1
BIN	1	1	0	0	...	1	1	...	1	1	1

- Grouping of Operators and Libc Functions (maybe represented differently by the compiler)
  - $<$, $\leq$, $>$, $\geq$ → comOpr
  - printf, puts → printf

P2. Control Block similarity

Solving Challenge 2 : Architecture & compiler variability

- Measures the similarity of *internal-branching blocks using feature-based vectors

  * Condition, Else-condition, Loop

Source code

```
int add(int x){ return x + 1; }
void main(int a, char *s){
 if (a >= 15){
  int res = strcmp(s, "foo");
  printf("%d%d", add(a), res);
 }
}
```

Key feature vector	Local var. used	Param. var. used	Opt. used		...	C library func. used		...	UDF count mtach	14 (0xe)	15
			Equ Opr	Com Opr		printf	strcmp				
SRC	0	1	0	1	...	0	0	...	1	0	1
BIN	0	1	0	1	...	0	0	...	1	1	0

➔ 0.75

Binary (GCC -O0, decompiled by Ghidra)

```
void main(int a, char *s){
 uint uVar1, uVar2;
 if (0xe < a) {
  uVar1 = strcmp(s, "foo");
  uVar2 = add(a);
  printf("%d%d",(ulong)uVar2,(ulong)uVar1);
 }
```

Block content vector	Local var. used	Param. var. used	Opt. used		...	C library func. used		...	UDF count match	"foo"	add
			Equ Opr	Com Opr		printf	strcmp				
SRC	1	1	0	0	...	1	1	...	1	1	1
BIN	1	1	0	0	...	1	1	...	1	1	1

➔ 1.0

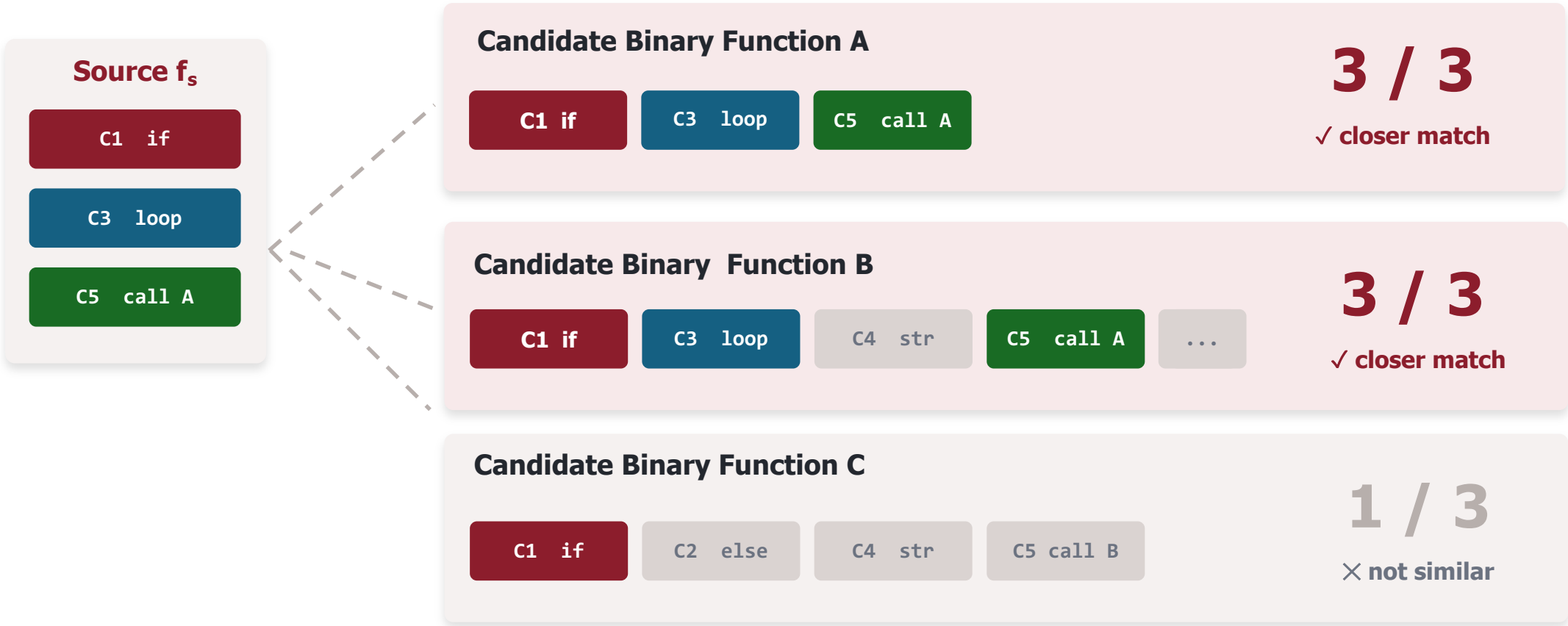
SBridge block similarity = 0.875

For comparison: token-cosine = 0.364, Levenshtein-based = 0.486

P3. Function matching

Solving Challenge 3 : Function inlining

- Match by how many of the source function's control blocks are found inside a candidate

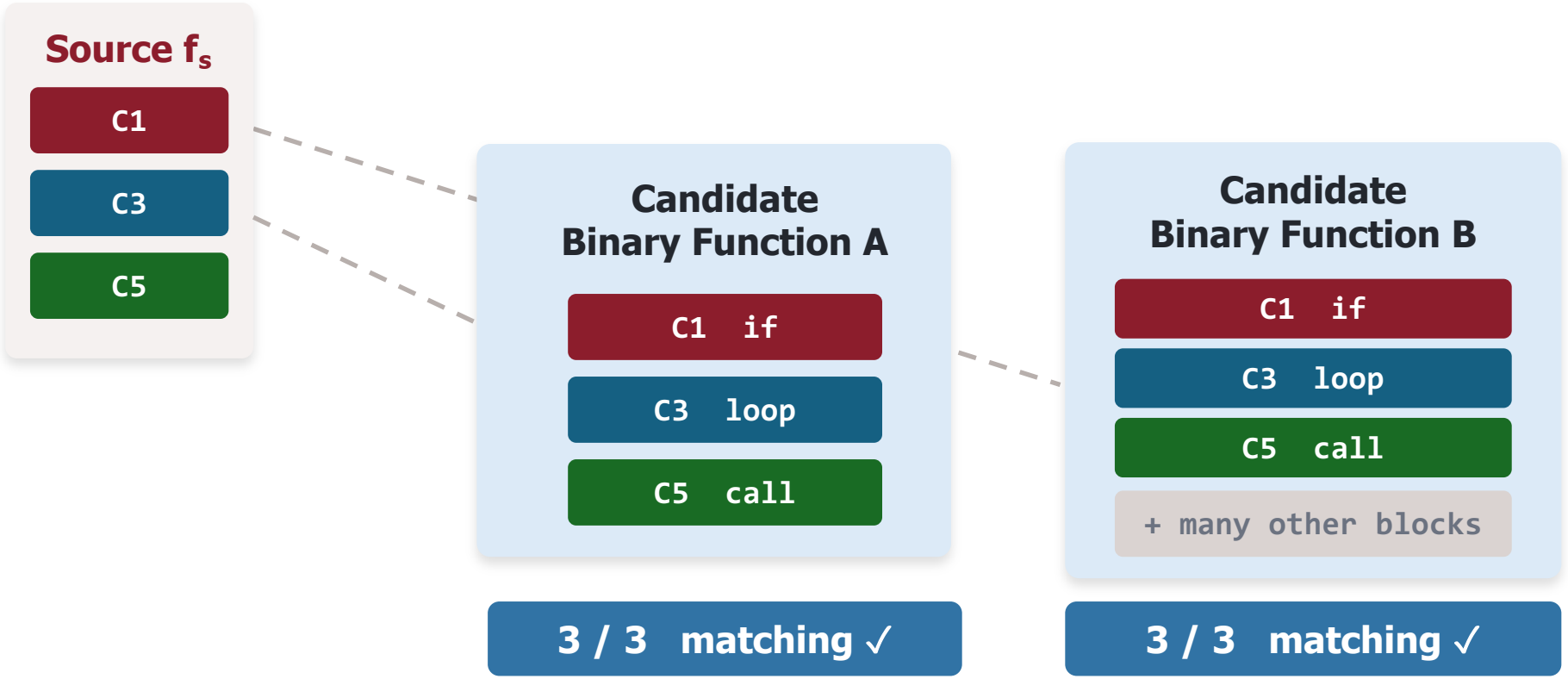


colored = block found in the source / grey = not matched

P3. Function matching

Solving Challenge 3 : Function inlining

- Control-block matching detects inlined functions

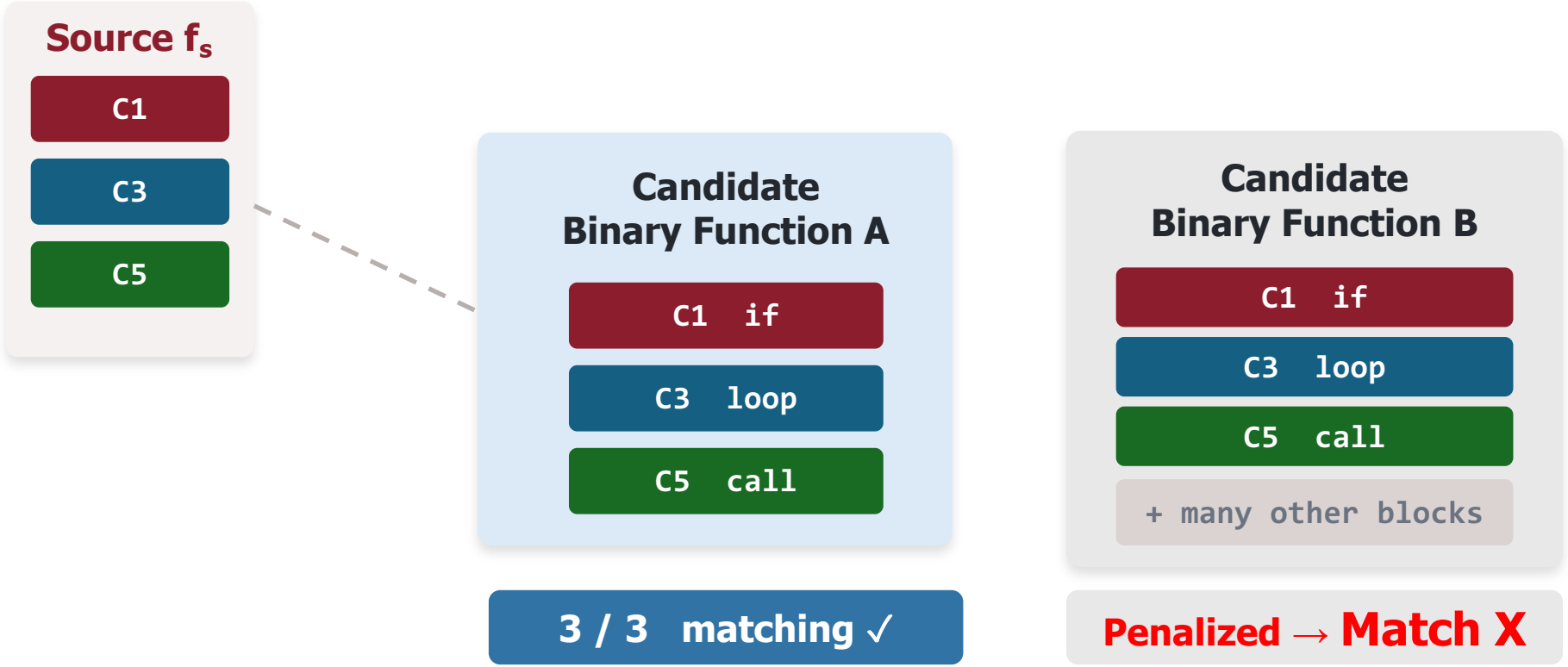


Potential false positives from tiny or coincidental matches

P3. Function matching

Solving Challenge 3 : Function inlining

- Length penalty when inlining is unlikely



No parent function match / no inline evidence → Apply length penalty → Low similarity

Our Approach : SBridge

Input : source function (query)

Source code function

SBridge

- ✓ C1 — context loss
- ✓ C2 — compiler / arch
- ✓ C3 — inlining

Output: ranked binary functions (Top 5)

FUN_1a0 ★ Top 1	similarity 0.94
FUN_2b4	0.41
FUN_3c8	0.27

Application : Software composition analysis, 1-day vulnerability detection

Evaluation : Dataset

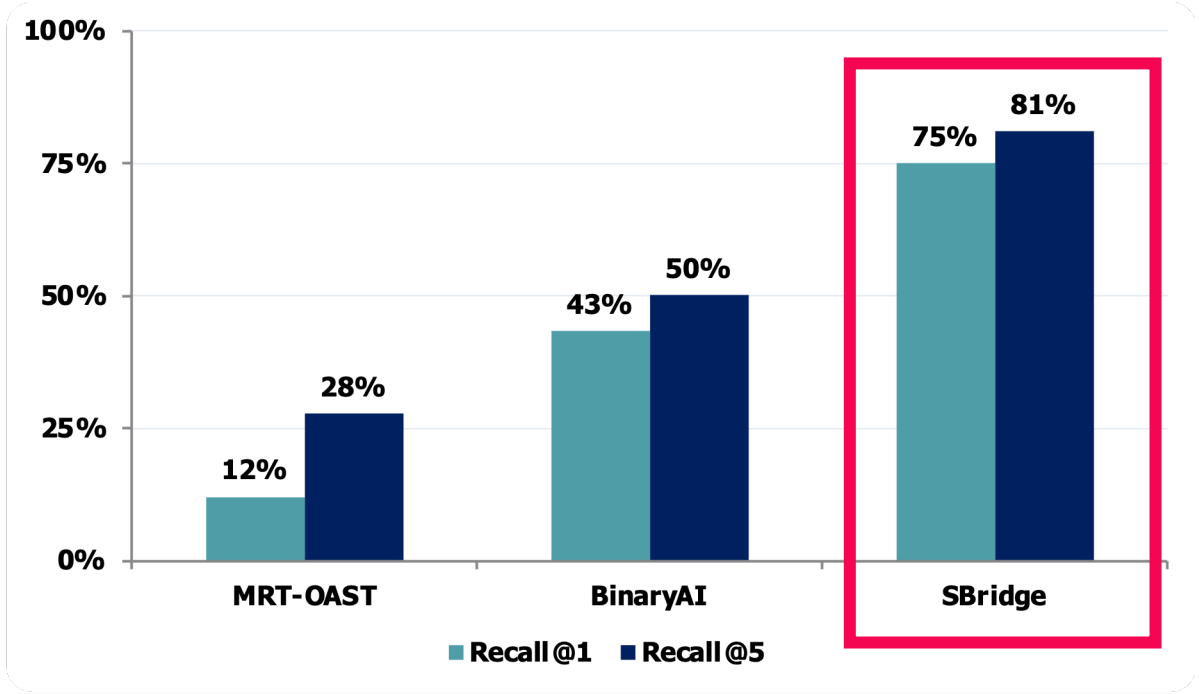
- **Packages: GNU Coreutils, Inetutils from BinKit**
- **Architectures: x86, x86-64, ARM32, ARM64**
- **Compilers: GCC 9.4.0, Clang 13.0**
- **Optimization and symbols : O0 / O2 and strip / no_strip**

  → **3,904 binaries containing 624,192 functions (Query: 1,618 source functions)**

- **Metrics: Recall@1, Recall@5, MRR**

RQ1 : Accuracy

How accurately does SBridge detect binary counterparts of source functions?



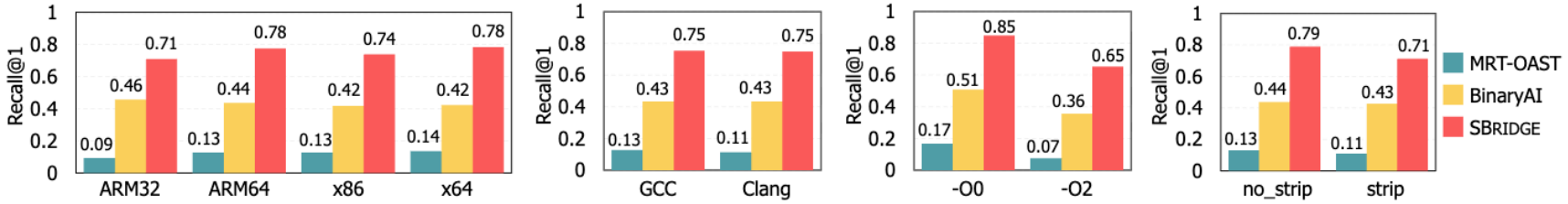
Overall Recall (avg. of all configs)

SBridge outperformed the MRT-OAST and BinaryAI

by achieving higher Recall@1 and Recall@5 in all configurations

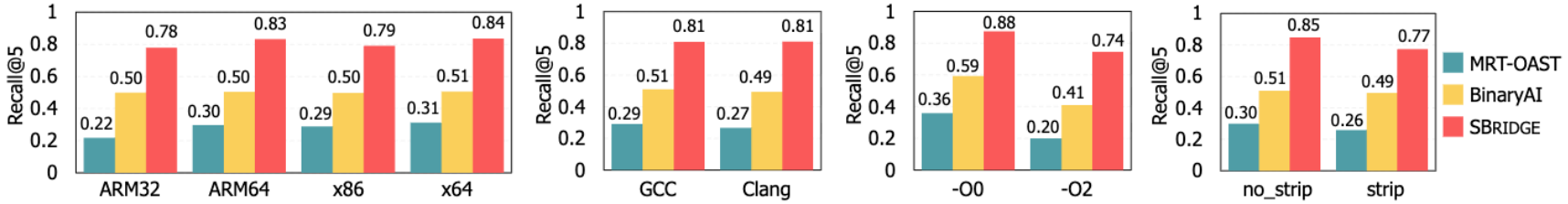
RQ1 : Accuracy

How accurately does SBridge detect binary counterparts of source functions?



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

Fig. 4. Recall@1 measurement results by architecture, compiler, optimization, and symbol management.



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

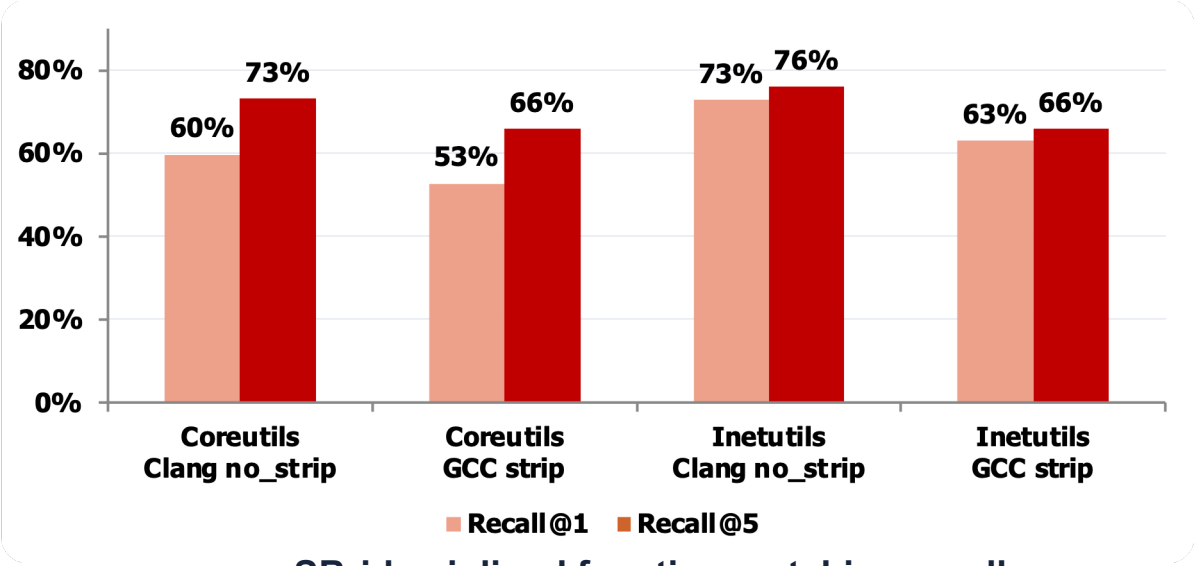
Fig. 5. Recall@5 measurement results by architecture, compiler, optimization, and symbol management.

SBridge achieved the highest Recall@1 and Recall@5 across architectures, compilers, optimization levels, and symbol settings

RQ2 : Efficacy (inlining)

How effectively does SBridge handle function inlining?

- **Only inlined functions Test**
  - SBridge: **73%** (Recall@1)
  - Previous Approaches: **1% ↓**



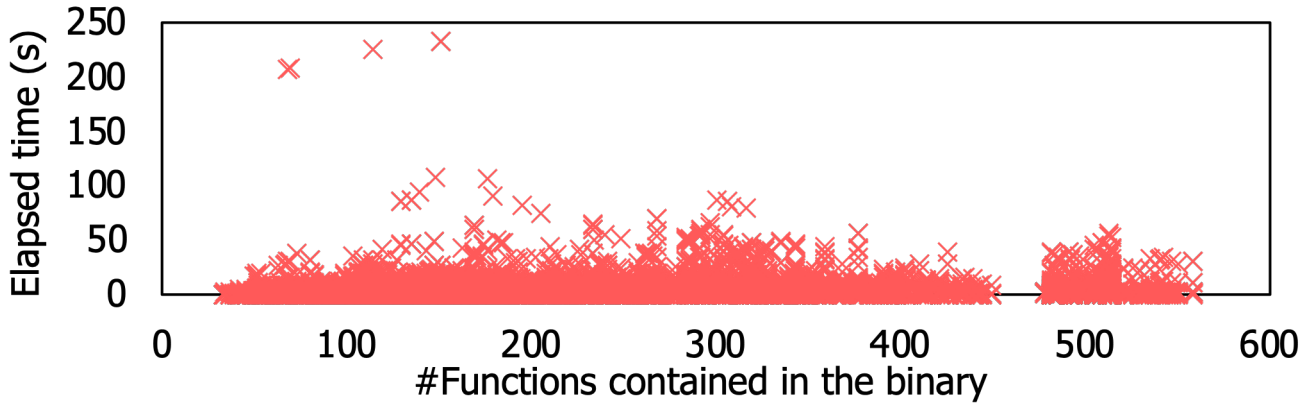
SBridge inlined function matching recall

SBridge can identify inlined functions,

whereas existing approaches struggle to detect

RQ3 : Performance

How fast and scalable is SBridge in detecting similar functions?



- Measures **matching time only**
- Excludes preprocessing by external tools (e.g., parsing and decompilation)
- **1.9 s** average per source-function query
- 96.8% of queries completed within 10 seconds

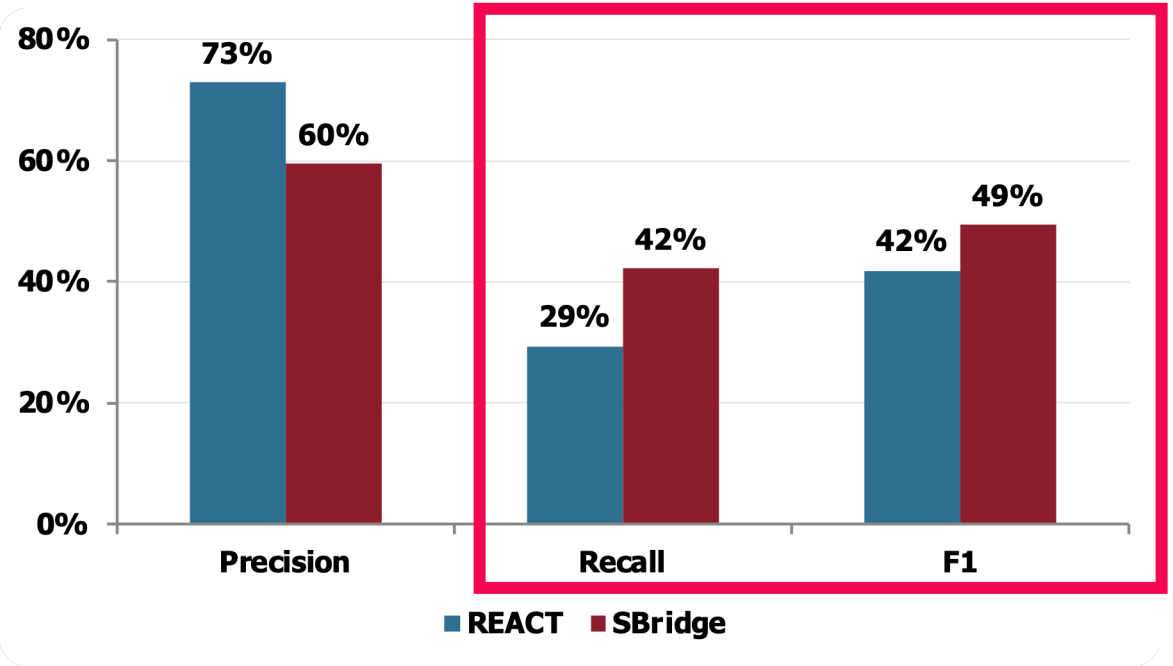
SBridge is practical for large-scale function matching

RQ4 : Application (1-day Vulnerability Detection)

How effective is SBridge when used for vulnerability detection?

- **Vulnerability detection(vs REACT) : 26 CVEs, 416 binaries**

  Each CVE in 8 configurations (2 compilers × 2 optimizations × 2 symbol management options)



Recall & F1 ↑

REACT relies on function names, making stripped binaries difficult to detection

Precision ↓

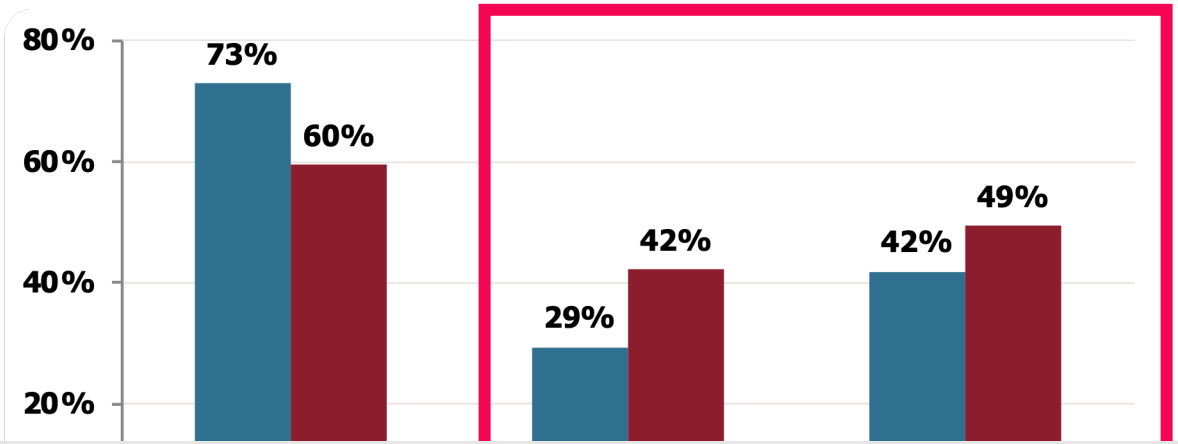
More false positives when patches modify only small code regions

RQ4 : Application (1-day Vulnerability Detection)

How effective is SBridge when used for vulnerability detection?

- **Vulnerability detection(vs REACT) : 26 CVEs, 416 binaries**

  Each CVE in 8 configurations (2 compilers × 2 optimizations × 2 symbol management options)



Recall & F1 ↑

REACT relies on function names, making stripped binaries difficult to detection

SBridge's approach for detecting similar functions across domains can also be effectively utilized for identifying propagated vulnerabilities

Conclusion

- Source and binary code are separated by a large gap (function inlining & stripped symbols)

- **SBridge** bridges this gap using **control block–based function matching**
  - **Outperforms existing approaches**
  - **Robust to function inlining** (achieving up to 73% Recall@1)
  - Efficient and scalable, requiring **1.9 s** per query on average

- **SBridge can also be utilized for real-world security analysis,**

  including vulnerability detection and binary software composition analysis (SCA)

Q & A

KOREA UNIVERSITY

SSP LAB
Software Security and Privacy Laboratory

Thank you

- SBridge Artifact: https://github.com/heedongy/SBridge_Artifact

CONTACT

- Heedong Yang (heedongy@korea.ac.kr, https://github.com/heedongy)
- Software Security & Privacy Lab @ Korea Univ. (https://ssp.korea.ac.kr , https://github.com/KUSSP)