Cross-Inlining Binary Function Similarity Detection

ICSE 2024

Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, Ting Liu
(Xi'an Jiaotong University)

Motivation

Software relies heavily on open-source code

- 96% of software contains open-source components; 84% has at least one vulnerability
- Reuse risks transfer to downstream binaries — must detect them automatically

Existing similarity detection assumes equal semantics

- Query function and target function are matched only when their full semantics align
- But **function inlining** mixes multiple functions into one — equality breaks down

Goal

- Systematically study **cross-inlining** binary function similarity detection

Problem: Function Inlining Breaks Matching

Motivating example (OpenSSL 1.0.1m, gcc-8.2.0, O3)

- The vulnerability is inherited, but existing tools score the pair **< 50% similarity** — vulnerability missed

Why exact matching fails

- Inlined binary functions are composed of **multiple source functions** mixed together
- Query function is only **part of** the target — partial-match problem, not equality

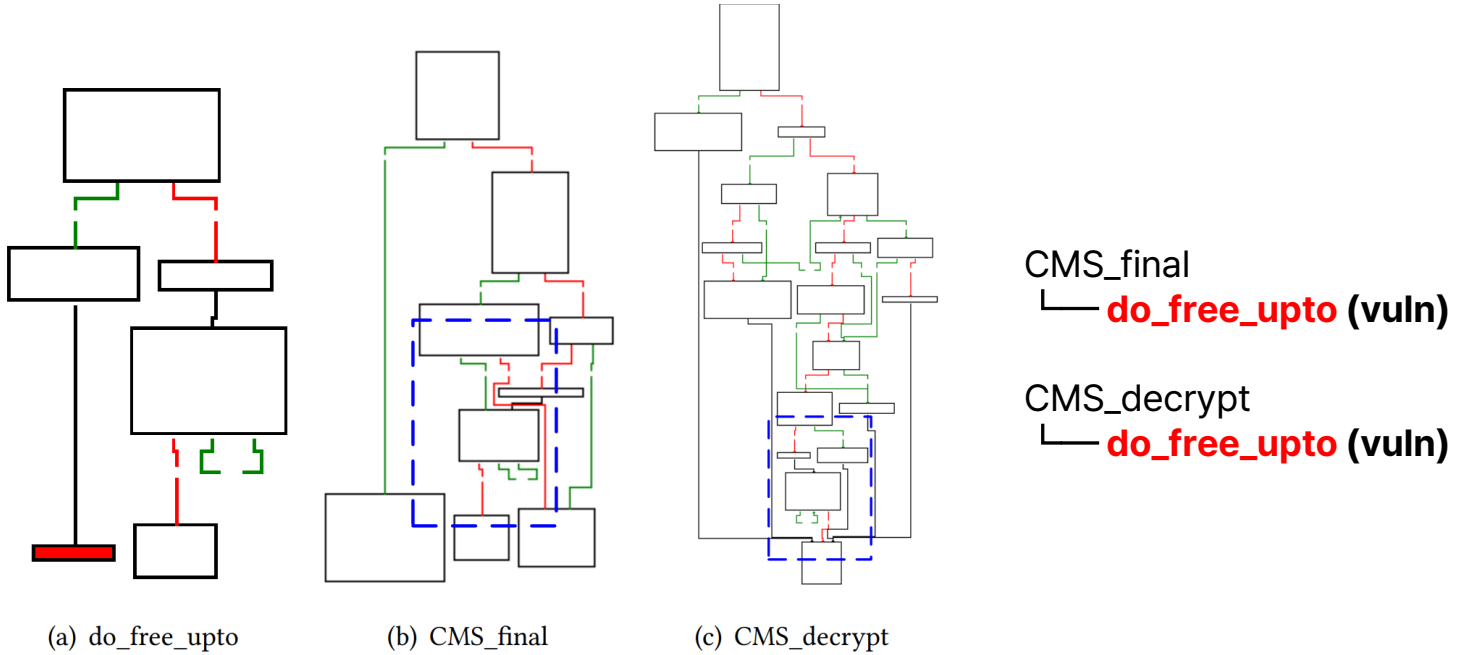


Figure 1: Cross-inlining matching example

Three Challenges

C1. Obscure binary semantics

- Binary semantics are **obscure and hard to separate**
- Once inlining happens, **the semantics of several functions are mixed**, making it harder to separate the inlined functions

C2. Different inlining contexts

- The same query can be inlined into many different callers, producing very different final functions
- e.g., *do_free_upto* is inlined into 5 different functions in a single binary

C3. Various inlining patterns

- A query can be **inlined into** others, or itself **inline** others — multiple structural patterns coexist

Cross-Inlining Dataset

Build two datasets

- Compile every project **twice** — once with inlining on, once with *fno-inline* — to obtain **Dataset-Inlining** and **Dataset-NoInlining**
- Same source, same compilers/optimizations/architectures → only inlining differs

Bridge function — the link across the two datasets

- Use *.debug_line* to recover binary→source function mappings
- **Bridge function** = a source function mapped to a binary function in **both** datasets — proves they share the same code
- e.g., source *do_free_upto* bridges binary *do_free_upto* (NoInlining) ↔ binary *CMS_decrypt* (Inlining)

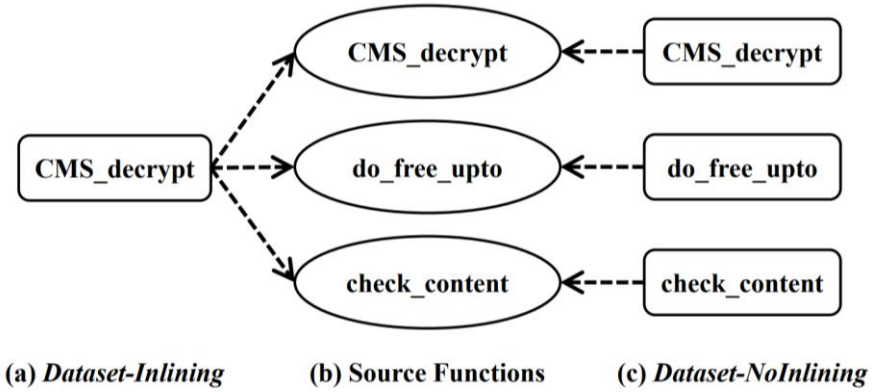


Figure 2: Example of constructing cross-inlining mappings

: binary function : source function

Three Cross-Inlining Patterns

Patterns are defined by where the **bridge function** sits in the source **f**unction **c**all **g**raph (FCG)

Leaf-Inlining

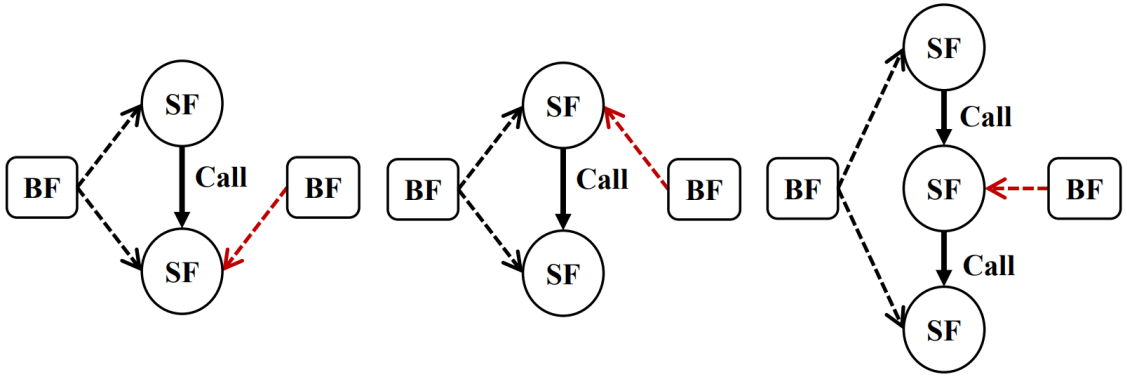
- Bridge is inlined into a caller; structure is largely preserved

Root-Inlining

- Bridge inlines other source functions; its content is split and interleaved

Internal-Inlining

- Bridge both inlines and is inlined — content is split and distributed across the binary



(a) Leaf-Inlining (b) Root-Inlining (c) Internal-Inlining

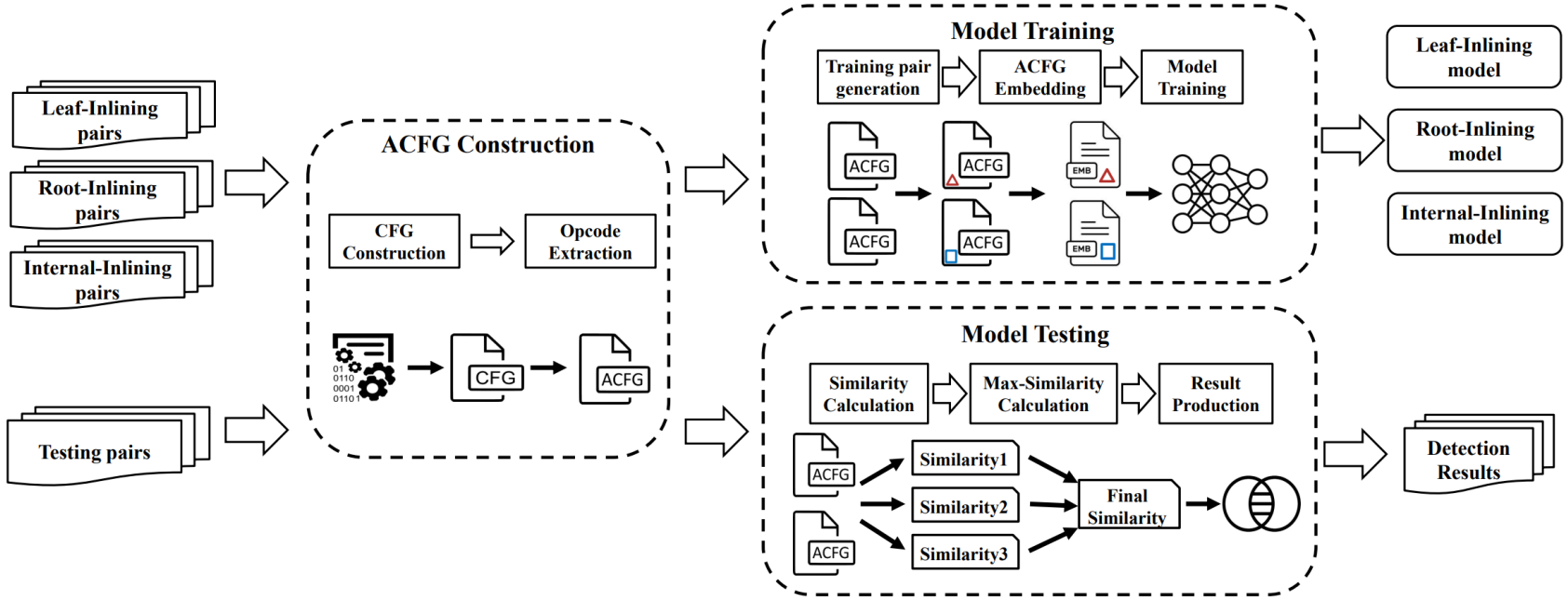
Figure 3: Cross-inlining matching patterns

: binary function : source function

CI-Detector: Overview

A pattern-based detector for cross-inlining matching

- **Step 1.** Build **ACFG** (Attributed CFG) for every binary function
- **Step 2.** Train **three GNN models**, one per pattern (Leaf / Root / Internal)
- **Step 3.** Score a test pair with all three models, then **aggregate** the similarities into a final score



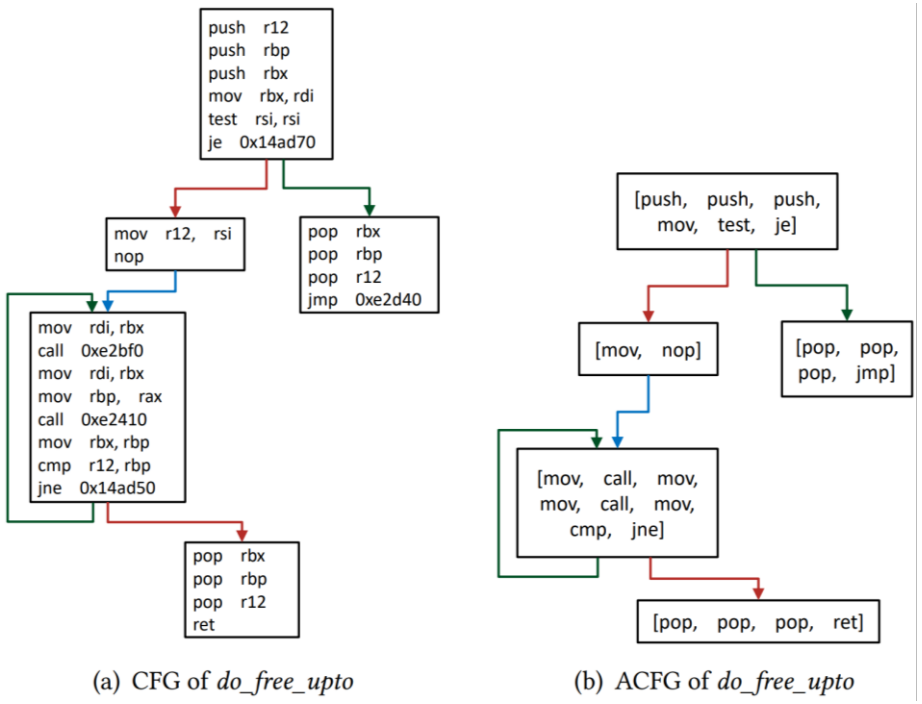
ACFG: Attributed Control Flow Graph

Definition

- CFG of basic blocks, where each node is annotated with the **sequence of opcodes** in that block

Why opcodes only?

- Operands — registers, immediates, addresses — shift across inlining contexts and are **unstable**
- Opcodes (e.g., *mov, test, je, call*) stay similar even after inlining — robust signal



(a) CFG of *do_free_upto* (b) ACFG of *do_free_upto*

Model Training & Testing

Embed each ACFG into a vector

- Node feature: opcode **bag-of-words** per basic block
- Graph encoder
    - a neural network with a similar architecture to GMN · encoder (MLP)

Train one model per inlining pattern

- **Pairs by bridge function**
    - positive = (equal binary, cross-inlining binary) sharing the same bridge
    - negative = a binary that does not share that bridge

Score a test pair → take the maximum

- Run the pair through all 3 models → 3 similarities → final = **max**

Experimental Setup

Dataset

- 9 compilers × 4 optimizations × 6 architectures **= 216 combinations**
- 51 projects (coreutils, binutils, ...) → **5,621,140 cross-inlining pairs**

Train / Test Split

- Project-level **80 / 10 / 10** (train / val / test
- Test set: 40,000 positive + 40,000 negative pairs — same pairs used to evaluate baselines

Model & Tools

- threshold **0.55** (best F1 on training pairs)
- IDA Pro 7.3, Capstone

Research Question 1, 2

Table 2: Effectiveness of CI-Detector and existing works

Method	Pattern	Accuracy	Precision	Recall	F1	AUC
Gemini	Leaf	0.50	0.50	1.00	0.67	0.60
	Root	0.71	0.69	0.77	0.73	0.79
	Internal	0.73	0.70	0.80	0.75	0.57
Safe	Leaf	0.53	0.53	0.53	0.53	0.54
	Root	0.67	0.62	0.85	0.72	0.74
	Internal	0.52	0.52	0.71	0.60	0.55
GMN	Leaf	0.56	0.56	0.56	0.56	0.59
	Root	0.74	0.71	0.80	0.75	0.83
	Internal	0.57	0.57	0.57	0.57	0.61
Bingo	Leaf	0.71	0.67	0.82	0.74	0.73
	Root	0.75	0.68	0.95	0.79	0.80
	Internal	0.69	0.66	0.79	0.72	0.72
Asm2Vec	Leaf	0.50	0.50	1.00	0.67	0.54
	Root	0.50	0.50	1.00	0.67	0.55
	Internal	0.50	0.50	1.00	0.67	0.54
CI-Detector	Leaf	0.87	0.81	0.97	0.88	0.89
	Root	0.88	0.81	0.99	0.89	0.87
	Internal	0.87	0.81	0.96	0.88	0.88

RQ3: Efficiency

Testing time per pair

- **Safe** < 0.001s — self-attentive neural network only
- **Gemini, GMN** ~ 0.005s — single GNN model
- **CI-Detector** ~ 0.013s — runs three GNN models

Still practical

- Slower than baselines, but processes **200,000 pairs in ~2/3 hour**

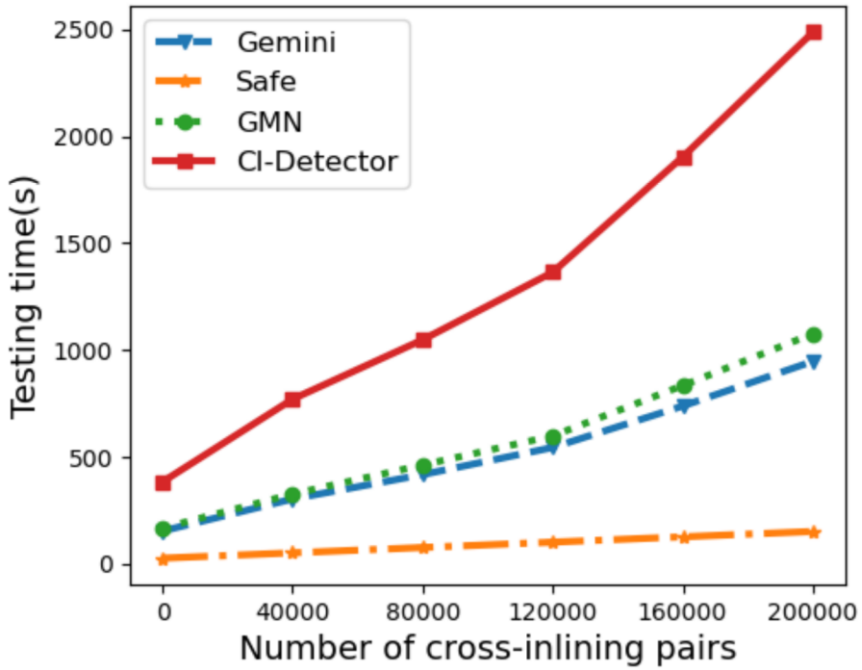


Figure 9: Testing time of CI-Detector

Conclusion

CI-Detector

- Defined cross-inlining as a partial-match problem
- 3 patterns: Leaf / Root / Internal

Limitations (My thoughts)

- Does not consider cross-optimization, cross-compiler, or cross-architecture settings
- Weak on large function-size mismatch
    - e.g., 6 blocks vs. 857 blocks → match fails

Thank You