Cross-Inlining Binary Function Similarity Detection

Ang Jia
jiaang@stu.xjtu.edu.cn
Xi'an Jiaotong University
China

Ming Fan*
mingfan@mail.xjtu.edu.cn
Xi'an Jiaotong University
China

Xi Xu
xx19960325@stu.xjtu.edu.cn
Xi'an Jiaotong University
China

Wuxia Jin
jinwuxia@mail.xjtu.edu.cn
Xi'an Jiaotong University
China

Haijun Wang
haijunwang@xjtu.edu.cn
Xi'an Jiaotong University
China

Ting Liu
tingliu@mail.xjtu.edu.cn
Xi'an Jiaotong University
China

ABSTRACT

Binary function similarity detection plays an important role in a wide range of security applications. Existing works usually assume that the query function and target function share equal semantics and compare their full semantics to obtain the similarity. However, we find that the function mapping is more complex, especially when function inlining happens.

In this paper, we will systematically investigate cross-inlining binary function similarity detection. We first construct a cross-inlining dataset by compiling 51 projects using 9 compilers, with 4 optimizations, to 6 architectures, with 2 inlining flags, which results in two datasets both with 216 combinations. Then we construct the cross-inlining function mappings by linking the common source functions in these two datasets. Through analysis of this dataset, we find that three cross-inlining patterns widely exist while existing work suffers when detecting cross-inlining binary function similarity. Next, we propose a pattern-based model named CI-Detector for cross-inlining matching. CI-Detector uses the attributed CFG to represent the semantics of binary functions and GNN to embed binary functions into vectors. CI-Detector respectively trains a model for these three cross-inlining patterns. Finally, the testing pairs are input to these three models and all the produced similarities are aggregated to produce the final similarity. We conduct several experiments to evaluate CI-Detector. Results show that CI-Detector can detect cross-inlining pairs with a precision of 81% and a recall of 97%, which exceeds all state-of-the-art works.

CCS CONCEPTS

• **Software and its engineering → Software reverse engineering**; **Search-based software engineering**.

KEYWORDS

Cross-Inlining, Binary Similarity Detection, Inlining Pattern

*Corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639080

ACM Reference Format:
Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. 2024. Cross-Inlining Binary Function Similarity Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639080

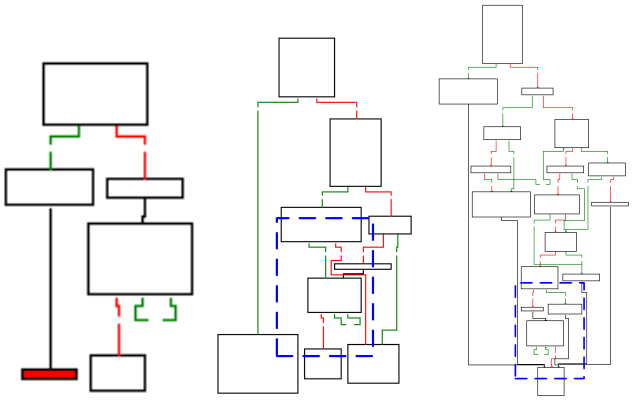
1 INTRODUCTION

Most software today is not developed entirely from scratch. Instead, developers rely on a range of open-source components to create their applications [56]. According to a recent report [6], 96% of the software contains open-source code. Although using open-source components helps to finish projects quicker and reduce costs, improper reuse introduces security and legal risks [40]. Of the 1,703 codebases scanned in 2022, 87% include security and operational risk assessments [6], where 54% of software has license conflicts and 84% of software contains at least one vulnerability. To make it worse, downstream software often relies on close-sourced third-party libraries [7]. Security and legal risks hidden in the binaries generated by the upstream supplier may be unintentionally transferred to the downstream developers or end-users.

To help resolve these code-reuse-related issues, many binary code similarity analysis works are proposed and have been applied in various applications, including code search [29, 30, 37, 63, 79, 81], OSS reuse detection [19, 31, 35, 36, 43, 53, 68, 71, 77], vulnerability detection [25, 26, 38, 76] and patch presence test [51, 61, 74, 78, 82]. They usually regard the vulnerable functions or reused functions as the query functions and the functions in the commercial software as the target functions and produce the detection results by calculating the similarity between query functions and target functions.

To obtain function-to-function similarities, most existing binary similarity analysis works usually assume that the target function shares the same semantics with the query function and try to find exact matches between them. However, we discover that the query function and the target function do not always share equal semantics, especially when function inlining happens.

Figure 1 shows a vulnerable binary function *do_free_upto* and two binary functions *CMS_final* and *CMS_decrypt* in OpenSSL 1.0.1m [15] compiled using gcc-8.2.0 with O3. Figure 1(a) shows the CFG (control flow graph) of the vulnerable function *do_free_upto*. *do_free_upto* is associated with the CVE-2015-1792 [14] which lacks an examination of null value that allows remote attackers to cause a denial of service. Figure 1(b) and Figure 1(c) respectively presents CFG of function *CMS_final* and *CMS_decrypt*. From the debug information, we notice that function *CMS_final* and *CMS_decrypt*



(a) do_free_upto (b) CMS_final (c) CMS_decrypt

Figure 1: Cross-inlining matching example

both inline the vulnerable function *do_free_upto*, copy the function body of *do_free_upto* (we use the blue dotted rectangle to represent it), and thus inherit the vulnerability.

However, when we try to use the existing works [25, 26, 38, 76] to match the vulnerable function *do_free_upto* with the function *CMS_final* and *CMS_decrypt*, the returned similarities are all less than 50%. We notice that though functions *CMS_final* and *CMS_decrypt* are vulnerable, they also contain function contents from their own and other inlined functions. For example, binary function *CMS_final* is compiled from source function *CMS_final* and *do_free_upto*, while binary function *CMS_decrypt* is compiled by source function *CMS_decrypt*, *check_content* and *do_free_upto*. The vulnerable function is only part of the function *CMS_final* and *CMS_decrypt*. Thus, it is difficult to identify the vulnerable function through exact matching.

Function inlining widely exists in binaries [48]. Though many works [25, 26, 29, 30, 37, 38, 51, 61, 63, 74, 76–79, 81, 82] are proposed to resolve the cross-optimization, cross-compiler, and cross-architecture problems in binary code similarity, few works are targeted at cross-inlining. There are still three challenges for resolving the cross-inlining binary function similarity detection.

C1: Obscure binary semantics. Different from source code which has rich semantics and is easy to understand, the semantics in binaries is obscure and hard to separate. Moreover, when function inlining happens, the semantics of several functions are mixed, which makes it harder to separate the inlined functions.

C2: Different inlining contexts. The query functions (vulnerable function or reused function) can be called and inlined by different functions. For example, the vulnerable function *do_free_upto* has been inlined into five different functions in one single binary. The different contexts around the inlined functions lead to different compositions of final functions.

C3: Various inlining patterns. Apart from the cases in which a vulnerable function is inlined into other functions, we also noticed that a vulnerable function can also inline other functions. For example, the vulnerable function *ssl23_get_client_hello* in OpenSSL 1.0.1j, associated with CVE-2014-3569 [13], has inlined another normal function *ssl2_get_server_method* when compiling using gcc-8.2.0 with O3. The various inlining patterns make cross-inlining detection a more complicated problem.

To tackle the challenges that function inlining brings, in this paper, we propose the first study to systematically investigate the solution of cross-inlining binary function similarity detection.

To tackle **C1**, we combine the opcodes and CFG to form ACFG (Attributed CFG) to represent the function semantics. As shown in Figure 1, though the vulnerable function is inlined into other functions, the vulnerable part has a similar structure and similar code logic.

To tackle **C2**, we use GNN [57] to learn the similarity of cross-inlining function pairs. GNN can learn from local structures of ACFG and thus identify the similar part of cross-inlining function pairs.

To tackle **C3**, we first conduct an empirical study and summarize three kinds of inlining patterns. Then, we train separate models for each inlining pattern to calculate cross-inlining similarities. Finally, we aggregate the similarities from these models to obtain the final detection results.

We evaluate our method on a cross-inlining dataset and results show that our method can obtain an AUC of 90.62% and exceed all state-of-the-art methods.

We summarize our contribution as follows:

- We create a cross-inlining dataset and we propose a labeling method to identify cross-inlining function pairs.
- We conduct an empirical study on the cross-inlining dataset and we summarize three inlining patterns that help better learn the cross-inlining similarities.
- Based on the three patterns, we propose a pattern-based ensemble method named CI-Detector. CI-Detector can detect cross-inlining pairs with a precision of 81% and a recall of 97%, which exceeds all state-of-the-art works.

To facilitate further research, we have made the source code and dataset publicly available [16].

2 BACKGROUND

2.1 Binary Function Similarity Detection

There are two major ways to detect binary code similarity, dynamic and static approaches. Dynamic approaches [34, 44, 47, 52, 55, 59, 65–67, 75], which obtain the program semantic by executing the program with test input, usually have a high overhead and suffer from low coverage. Static approaches [25, 26, 29, 30, 37, 38, 63, 76, 79, 81], which directly extract features from binary executables, are more scalable to large amounts of binaries. However, under different compilation settings, the same source code can be compiled into binaries with different syntax, different layouts, and different instruction sets, which introduces cross-optimization, cross-compiler, and cross-architecture binary similarity detection approaches.

Cross-optimization binary similarity detection works [20, 23–26, 30, 37, 38, 60, 63, 72, 76, 84] aim to detect similarities of binary functions compiled by same source functions but with different optimizations. For example, Gitz [24] decomposes binary procedures to comparable fragments and uses the compiler to re-optimize them to obtain canonical strands for comparison. Though different optimizations lead to syntactically different binary codes, re-optimizing them with high optimizations converts them into syntactically similar binaries.

Cross-compiler detection works [20, 30, 37, 60, 63, 76] aim to detect similarities of binary functions compiled by different compilers. For example, Asm2Vec [30] learns the lexical semantic relationships and constructs the function embedding by aggregating instruction embeddings. Different compilers use different conventions to arrange the binary code while existing works learn the semantics from adjacent instructions and aggregate them to detect similar functions.

Cross-architecture detection works [20, 24, 37, 38, 60, 63, 72, 76, 84] aim to detect similarities of binary functions compiled to different architectures. For example, Gemini [76] constructs ACFG (Attributed Control Flow Graph) and leverages graph embedding to generate binary function embeddings. Though different instruction sets are used in different architectures, the control flow structure can still preserve similar execution logic in binary code.

Though existing works have paid enormous effort to cross-optimization, cross-compiler, and cross-architecture problems, few works have systematically explored cross-inlining binary function similarity detection. Some works including Bingo [20] and Asm2Vec [30] have proposed manually designed rules to resolve some issues under function inlining, but they cannot cover the various inlining patterns and cannot provide a complete view of the cross-inlining binary function similarity detection.

2.2 Function Inlining

Function inlining, or inline expansion, is a manual or compiler optimization that replaces a function call site with the body of the called function [5]. Though the direct effect of function inlining is to eliminate call overhead, its primary benefit is to allow further optimizations. As the contents of multiple functions are aggregated together, more intra-procedural optimizations can be applied without requiring inter-procedural optimizations.

Though inlining brings improvement to the executable performance, it also increases code size. Thus, it is a trade-off to determine when to inline for a balance between these benefits and costs. A range of different heuristics [18, 21, 27, 28, 33, 45, 46, 83] have been explored for inlining. Usually, an inlining algorithm has a certain code budget (an allowed increase in program size) and aims to inline the most valuable call sites without exceeding that budget.

Once the compiler has decided to inline a particular function, performing the inlining operation itself is usually simple. Depending on whether the compiler inlines functions across code in different languages, the compiler can do inlining on either a high-level intermediate representation (like abstract syntax trees) or a low-level intermediate representation. In either case, the compiler simply computes the arguments, stores them in variables corresponding to the function's arguments, and then inserts the body of the function at the call site [5].

2.3 Cross-Inlining Binary Similarity Detection

Existing works have taken preliminary research of binary code similarity analysis under function inlining. The first binary function similarity detection work that takes function inlining into consideration is Bingo [20]. Bingo summarizes several patterns that a caller function should inline its callee functions, and conducts inlining to simulate the functions generated by function inlining. Asm2Vec [30] also uses Bingo's strategies to tackle function inlining. However, inlining the callee function into caller functions requires that the query function and the target function should have the same context in FCG (Function Call Graph). It also raises a higher cost to compare functions with bigger function contents.

Jia [49] has conducted the first systematical empirical study to investigate the effect of function inlining on binary function similarity analysis. They evaluated four works on the dataset with inlining, and results show that most works suffer a 30%-40% performance loss when detecting the functions with inlining. Besides, most works ignored the inlined functions, causing functions that have inlined vulnerable functions to be undetected. Our work also noticed these two issues, and we propose a method that can tackle these two challenges.

Recently, Jia [50] has proposed a method named O2NMatcher to investigate the binary2source matching method under function inlining. In binary2source matching, binary functions can be generated by several source functions. O2NMatcher tries to find the source function sets as the matching targets of binary functions with inlining. That requires the context of the source functions. In this paper, we will propose a more scalable method without requiring the context of query functions and target functions. Besides, we will introduce two additional cross-inlining patterns in this work.

3 EMPIRICAL STUDY

In this section, we will conduct an empirical study to investigate cross-inlining binary function similarity detection tasks.

3.1 Problem Definition

We first define the objective of cross-inlining binary function similarity detection as follows:

Given a query function q without inlining and a target function t with inlining, in binary form, the cross-inlining binary function similarity detection aims to detect whether function q is inlined into the function t.

In this paper, we take the first step in cross-inlining detection — we study the cross-inlining problems without introducing cross-optimization, cross-compiler, and cross-architecture problems. That is the query function q and the target function t are compiled by the same compilers using the same optimizations to the same architectures. However, the query function q and the target function t can still be compiled under multiple optimizations, compilers, and architecture.

3.2 Cross-inlining Dataset Construction

We construct a cross-inlining dataset by using Binkit [54] and propose a cross-inlining labeling method leveraging the function inlining identification method in Jia [49].

Binkit is a binary code similarity analysis benchmark. We use it to construct two datasets: *Dataset-Inlining* and *Dataset-NoInlining*. Both datasets are constructed by compiling 51 gnu projects [3] using 9 compilers (GCC v4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0 and Clang v4.0, 5.0, 6.0, 7.0), 4 optimizations (O0, O1, O2, O3), to 6 architectures (ARM32, ARM64, MIPS32, MIPS64, X86-32. X86-64), resulting in total 216 combinations. *Dataset-NoInlining* is compiled with an additional flag *"fno-inline"* to turn off the function inlining. Finally,

Dataset-Inlining is composed of 50,760 binaries and 12,644,259 binary functions, where 2,195,920 (17.4%) are binary functions with inlining. *Dataset-NoInlining* is composed of 50,760 binaries and 15,233,501 binary functions.

We further construct the binary2source function mappings for these two datasets respectively. We follow the workflow in Jia [49]: we first extract the address-to-line mappings from the *.debug_line* section, then we extract the address-to-binary-function and line-to-source-function relation to get the functions that the address or line belongs to, and finally we construct the binary2source function mappings by aligning the binary functions with their mapped source functions.

Using the binary2source function mappings, we can identify binary functions with inlining by the number of source functions they map. If a binary function maps to more than one source function, it will be considered a function with inlining. In *Dataset-NoInlining*, the binary functions are mostly mapped to one source function. Note that these are binary functions with inlining in *Dataset-NoInlining*, as *"fno-inline"* cannot prevent user-forced inlining such as "*always_inline*". We exclude these binary functions with inlining from *Dataset-NoInlining*.

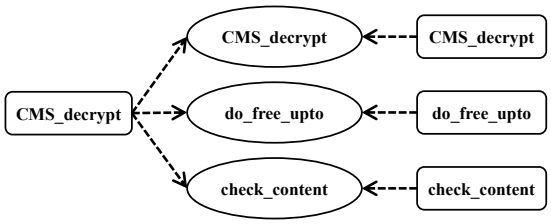


Figure 2: Example of constructing cross-inlining mappings

Then we use the source function as the bridge to construct cross-inlining function mappings. Figure 2 presents an example. We use rounded rectangles to represent binary functions and ellipses to represent source functions. The dotted arrows represent the binary2source function mappings. After constructing binary2source function mappings, we notice that binary function *CMS_decrypt* in *Dataset-Inlining* maps to source functions *CMS_decrypt*, *do_free_upto*, and *check_content*. The binary function *do_free_upto* in *Dataset-NoInlining* maps to source function *do_free_upto*. Here, we regard the source function *do_free_upto* as a *bridge function*. Leveraging the bridge function, we can obtain a cross-inlining function pair: *CMS_decrypt* in *Dataset-Inlining* and *do_free_upto* in *Dataset-NoInlining*.

By traversing all binary functions in *Dataset-Inlining* and *Dataset-NoInlining*, we construct 5,621,140 cross-inlining function pairs.

3.3 Cross-Inlining Patterns Analysis

After constructing the cross-inlining function mappings, we further investigate the cross-inlining patterns and their existence in the cross-inlining dataset. Firstly, we summarize three cross-inlining patterns according to the location of the bridge function in the FCG.

Figure 3 shows three cross-inlining patterns. We use rounded rectangles to represent binary functions (BF) and circles to represent source functions (SF). We use dotted arrows to represent

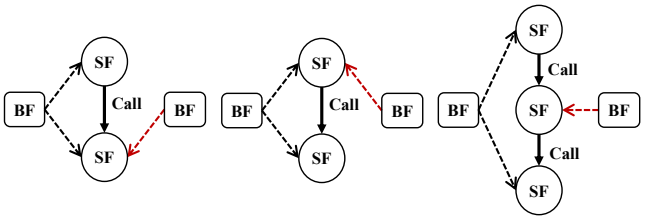


Figure 3: Cross-inlining matching patterns

binary2source function mappings and solid arrows to represent function calls between source functions. We summarize three cross-inlining patterns including *Leaf-Inlining*, *Root-Inlining*, and *Internal-Inlining*, respectively corresponding to the leaf node, the root node, and the internal node that the bridge function is in the source FCG.

As shown in Figure 3(a), when the bridge function is a leaf node, we classify this cross-inlining pattern as *Leaf-Inlining*. It is the same as the case in Figure 1. When the bridge function is inlined to the binary function with inlining, its content is surrounded by the source function which calls it and its structure can be mainly reserved.

When the bridge function is a root node as shown in Figure 3(b), the bridge function will inline other source functions into the body. We classify it as *Root-Inlining*. As a result, though the binary function with inlining still starts with the content of the bridge function, the content of the bridge function is split into several parts and filled in the content of inlined functions.

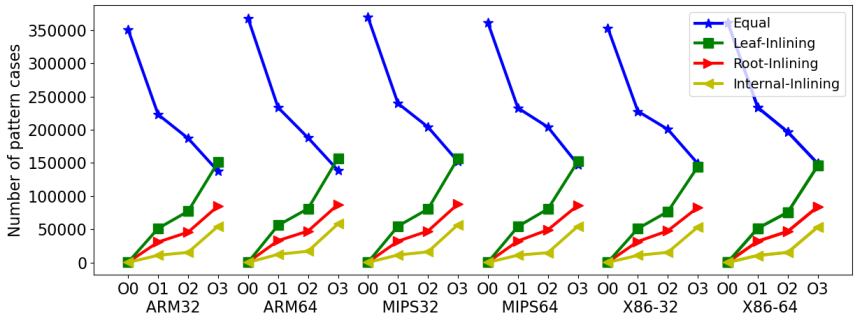
When the bridge function is an internal node of the source function as shown in Figure 3(c), the bridge function will not only be inlined but also inline other source functions. We classify it as *Internal-Inlining*. As a result, the function content is split and distributed in the binary functions with inlining.

We further summarize the distribution of these three cross-inlining patterns along with the traditional *Equal* pattern in *Dataset-Inlining* and *Dataset-NoInlining*. Since the statistics of the compilers in the same family present similar distribution [49], we aggregate the statistics of the same compiler families. As shown in Figure 4, we obtain the distribution of these patterns under 6 architectures, 2 compiler families, and 4 optimizations.

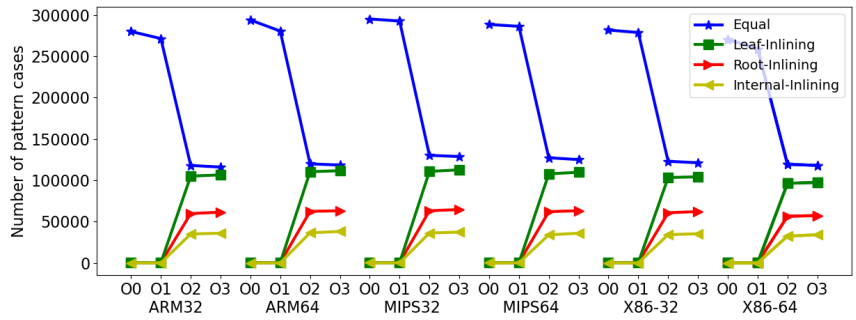
From Figure 4, we first find an increasing trend of cross-inlining patterns from O0 to O3. In detail, we observe that the *Leaf-Inlining* pattern is more common than *Root-Inlining* and *Internal-Inlining* patterns, and even exceeds the number of *Equal* pattern cases when applying O3. The large number of cross-inlining pattern cases indicates the importance of cross-inlining binary function similarity detection.

When comparing the cross-inlining patterns under GCC and Clang, we observe a different distribution. The number of cross-inlining cases keeps increasing from O0 to O3 under GCC, while it only increases from O1 to O2 under Clang. That is because GCC is increasingly adding inlining options to facilitate inlining, while Clang uses *"always inliner"* pass in O0 and O1, and *"inliner"* pass in O2 and O3.

The statistics of cross-inlining patterns under different architectures remain similar. The different architectures do not influence compilers on their inlining decisions.



(a) GCC



(b) Clang

Figure 4: Distribution of cross-inlining matching patterns

3.4 Cross-inlining Matching Evaluation

After analysis of these cross-inlining patterns, we want to evaluate existing works on matching these cross-inlining pairs. Here, we select three existing works, including Gemini [76], Safe [63], and GMN [57], to evaluate their performance on cross-inlining binary function similarity detection.

Gemini, Safe, and GMN are state-of-the-art works that have been compared in many research works [38, 62, 70, 80]. Gemini uses a GNN model called Structure2vec [22] with manually engineered features to compute function embeddings. SAFE uses the self-attentive sentence encoder [58] to learn cross-architecture function embeddings. GMN proposes a graph matching network with a cross-graph attention-based matching mechanism.

We use their released models and we run them to calculate the similarity of the cross-inlining pairs. In detail, for these three cross-inlining patterns, we first randomly select 40,000 positive pairs and 40,000 negative pairs from five projects (also testing projects in Section 5) in *Dataset-Inlining* and *Dataset-NoInlining* respectively. Then we run these models to calculate the pair similarities. Finally, we use Accuracy, Precision, Recall, F1-score, and AUC to evaluate the effectiveness of these models.

As the existing model only produces similarities between function pairs, we can only calculate their AUC before setting the threshold. Then, to calculate other metrics, we tried the threshold from 0.5 to 0.95 by the step of 0.05. When the similarity exceeds the threshold, we regard it as a positive pair, otherwise, a negative pair. Finally, we select the threshold that results in the best F1 and summarize the results as shown in Table 1.

Table 1: Effectiveness of existing works on cross-inlining

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

As shown in Table 1, we use "Leaf" as the abbreviation of "Leaf-Inlining", "Root" as the abbreviation of "Root-Inlining", and "Internal" as the abbreviation of "Internal-Inlining". We also use the same abbreviations in the rest tables.

In general, compared with their performance on equal pairs [62], existing works suffer a performance loss when detecting cross-inlining pairs. For example, Safe can achieve an AUC of more than 0.90 when detecting equal pairs [62], while it can only achieve an AUC of 0.54 on detecting *Leaf-Inlining* pairs, 0.74 on *Root-Inlining* pairs, and 0.55 on *Internal-Inlining* pairs. In the worst case, the best F1-score of Safe is only 0.53, while regarding all the detection pairs as positive pairs will obtain an F1-score of 0.67.

Existing works tend to perform better on *Root-Inlining* pairs compared with *Leaf-Inlining* and *Internal-Inlining* pairs. As shown in Table 1, Gemini can achieve an AUC of 0.79 on *Root-Inlining* pairs while it can only obtain an AUC of 0.60 and 0.57 on *Leaf-Inlining* and *Internal-Inlining* pairs. It can be attributed to inlining conventions. As small functions are more likely to be inlined, the bridge function

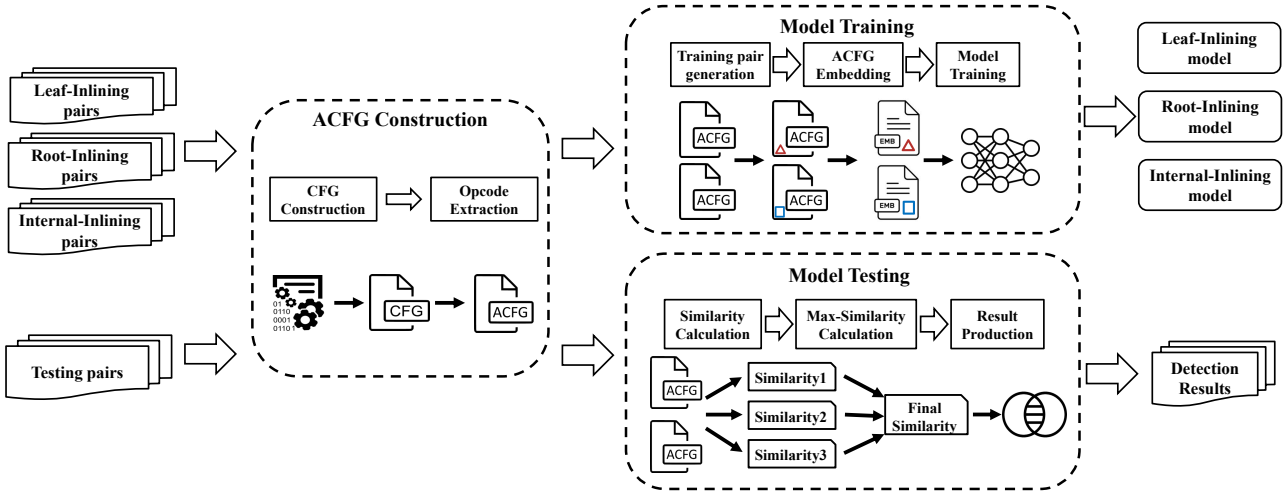


Figure 5: Overview of CI-Detector

still plays a major part in the binary function with inlining in the *Root-Inlining* pairs. However, in *Leaf-Inlining* and *Internal-Inlining* pairs, the bridge functions are those inlined functions that account for the small parts, making the detection more challenging.

Besides, GMN performs better on *Root-Inlining* pairs. For example, GMN can achieve an AUC of 0.83 when detecting *Root-Inlining* pairs, where it can obtain an F1-score of 0.75. GMN has also proven to be more effective in normal binary similarity detection [62]. In this work, we will use a neural network with a similar architecture to GMN to conduct cross-inlining binary similarity detection.

4 METHOD

In this section, we propose a method, named **CI-Detector** (**C**ross-**I**nlining **D**etector), for cross-inlining binary function similarity detection.

Figure 5 shows the overview of CI-Detector. In general, CI-Detector first extracts ACFGs for these cross-inlining pairs and trains three models for *Leaf-Inlining*, *Root-Inlining*, and *Internal-Inlining* patterns respectively. Then these models are used to calculate similarities for testing pairs and these similarities are aggregated to produce the final similarities.

In this section, we will introduce the ACFG Construction, Model Training, and Model Testing process of CI-Detector.

4.1 ACFG Construction

Accurate representation of function semantics is the basis of binary similarity detection. Structural representations, such as CFG, are considered stable representations whose structure varies little for similar code [42]. In this paper, we combine the CFG and the opcodes in the basic blocks to form the ACFG, as the representation of functions.

Figure 6 shows an example of constructing ACFG. Figure 6(a) shows the CFG of the function *do_free_upto*, we use rectangles to represent basic blocks and arrows to represent control flows. We also present the disassembled instructions in every basic block. An instruction consists of one opcode and zero or more operands. These operands can be registers, immediate values, or addresses.

Figure 6(c) shows the CFG of the function *CMS_final*. The function *do_free_upto* and the function *CMS_final* is a cross-inlining pair. The dotted rectangles in Figure 6(c) represent binary code from source function *CMS_final*, and rectangles with solid lines represent binary code from the bridge function *do_free_upto*.

When comparing the binary code in the cross-inlining pair compiled from the same bridge function, we noticed that the operands such as registers and addresses are different in these two functions. As the bridge function shares different contexts in these two binaries, function addresses have been changed and some registers have already been used in the binary function with inlining. Thus, these operands are not robust when conducting cross-inlining similarity detection. Instead, the opcodes in these basic blocks remain similar. Thus, we extract opcodes as attributes of nodes to form the ACFG.

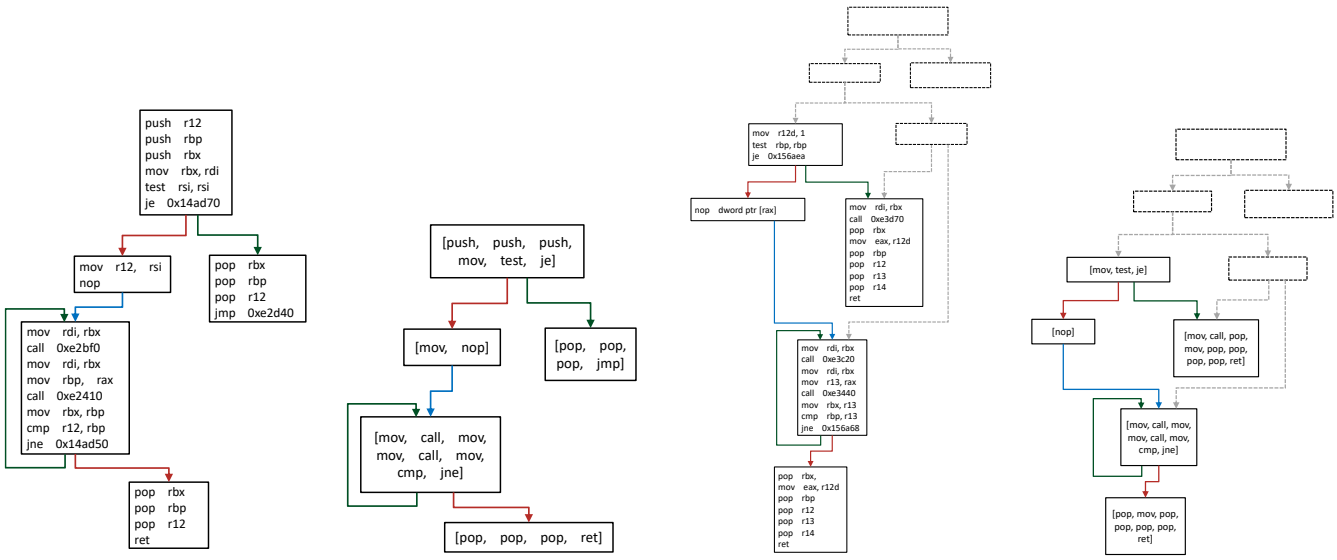
Figure 6(b) and Figure 6(d) are the corresponding ACFGs of function *do_free_upto* and function *CMS_final*. Comparing these two ACFGs, we noticed that the corresponding nodes share similar codes. Even for some nodes that have content from other source functions, such as the root node, they still share similar important opcodes (mov, test, je) that can help neural networks capture the similar semantics of the basic blocks.

4.2 Model Training

Considering the difference between these three cross-inlining patterns, we use GNN [57] to train separate models respectively for these patterns. In detail, we train three models including the *Leaf-Inlining* model, the *Root-Inlining* model, and the *Internal-Inlining* model respectively for *Leaf-Inlining* pairs, the *Root-Inlining* pairs and the *Internal-Inlining* pairs.

Training pair generation. In traditional binary code similarity detection, positive pairs are generated by selecting binary functions compiled from the same source function but compiled by different compilers, using different optimizations, to different architectures. These functions have the same function names, thus functions with the same names can be identified as positive pairs, and functions with different names are negative pairs [62].

However, it is more complex to construct positive and negative pairs in cross-inlining binary similarity detection. Functions with



(a) CFG of *do_free_upto* (b) ACFG of *do_free_upto* (c) CFG of *CMS_final* (d) ACFG of *CMS_final*

Figure 6: An example of ACFG construction

different names may not be negative pairs, and functions with the same name are not necessarily positive pairs (for example, equal pairs are not cross-inlining pairs). To construct positive and negative pairs for cross-inlining, we use the source bridge function as the key to construct a dict to arrange the cross-inlining relations.

Listing 1: An example of the cross-inlining relation

```
{
  "do_free_upto": {
    "equal": [
      "do_free_upto"
    ],
    "cross-inlining": [
      "CMS_final",
      "CMS_decrypt"
    ]
  }
}
```

Listing 1 shows an example of the cross-inlining relations. For the example in Figure 1, source function *do_free_upto* is the bridge function of two cross-inlining pairs, including function *CMS_final* with *do_free_upto*, and function *CMS_decrypt* with *do_free_upto*. In Listing 1, we construct a dict by setting the bridge function as the primary key. Then we set two secondary keys including "equal" and "cross-inlining". In this dict, the binary function *do_free_upto* is an "equal" value of the bridge function key *do_free_upto* while binary function *CMS_final* and *CMS_decrypt* are two "cross-inlining" values. Positive pairs can be generated by randomly selecting one function from the "equal" values and one function from "cross-inlining" values under the same source bridge function.

To generate the negative pairs, we randomly select a binary function with inlining that does not appear in the "cross-inlining" values of the current bridge function, and this binary function can be a negative example of the binary functions in the "equal" values.

ACFG Embedding. To convert ACFG to embeddings, we first represent the node features in vector form. Briefly, we use the bag-of-words model [64] to process these opcodes.

The bag-of-words model converts the opcodes into vectors by counting their frequencies. For example, the opcodes *[push, push, push, mov, test, je]* will be expressed as *{"push":3, "mov":1, "test":1, "je":1}*. When we define the key sequence as *[push, mov, test, je]*, then the vector of this node becomes *[3, 1, 1, 1]*.

Then, we convert the full graph into embedding using GNN [57]. The GNN embedding model comprises 3 parts: (1) an encoder, (2) propagation layers, and (3) an aggregator.

First, the encoder maps the node features to the initial node vectors through an MLP (Multi-layer Perceptron). Then, a propagation layer obtains new node representations by aggregating the representations of the node and its neighbors. Through multiple layers of propagation, the representation for each node will accumulate information in its local neighborhood. Finally, the aggregator will aggregate all the node representations to generate final graph representations. Through the above steps, we obtain the embeddings of ACFGs.

Model Training. For each function pair $(G1, G2)$ in the cross-inlining dataset, we can generate embeddings for their ACFGs. Then we use the Euclidean similarity to calculate their similarity and use the following margin-based pairwise loss to optimize the parameters:

$$Loss = \max\{0, \gamma - t(1 - d(G1, G1)\} \quad (1)$$

where $t \in \{-1, 1\}$ is the label for this pair, $\gamma > 0$ is a margin parameter, and $d(G1, G1)$ is the Euclidean distance. This loss encourage $d(G1, G1) < 1 - \gamma$ when the pair is similar ($t = 1$), and $d(G1, G1) > 1 + \gamma$ when $t = -1$. Then we use the gradient descent based algorithms to optimize the loss function.

Instead of training a single model on all cross-inlining patterns, we choose to train three models for the three cross-inlining patterns

respectively. Considering the difference between each pattern, the model can better fit the characteristics of a single pattern.

After respectively training on the *Leaf-Inlining* pairs, *Root-Inlining* pairs, and *Internal-Inlining* pairs, we obtained three models for these three cross-inlining patterns.

4.3 Model Testing

As shown in Figure 5, when inputting the testing pairs, we use the *Leaf-Inlining* model, *Root-Inlining* model, and *Internal-Inlining* model to produce three similarities for each testing pair. The similarity is calculated through the following equation:

$$sim(G1, G2) = \frac{1}{1 + d(G1, G2)} \tag{2}$$

Through Equation 2, we normalize the similarity to $(0, 1]$.

Then we use the maximum similarity as the final similarity of the testing pair. The design of similarity calculation is based on the following reasons. On the one hand, when the input pair is a positive pair, it will obtain at least one high similarity. Using the maximum similarity can help identify the positive pairs. On the other hand, when the input pair is a negative pair, these three models will produce low similarities. The maximum similarity will be low which indicates that it is a negative pair.

After obtaining the similarity, we still need a threshold to distinguish positive pairs and negative pairs. We try different thresholds in the training cross-inlining pairs and find that the threshold 0.55 produces the best F1 score. Then we use this threshold to compare with similarities of testing pairs and produce the labels for them.

5 EXPERIMENTS

In this section, we will first introduce the setup of our experiments. Then, we answer the following research questions to validate the performance of CI-Detector.

RQ 1. *Can CI-Detector effectively detect the cross-inlining binary code similarity?*

RQ 2. *How effective is CI-Detector, compared to existing works?*

RQ 3. *What is the contribution of each model in CI-Detector?*

RQ 4. *Can CI-Detector efficiently detect the cross-inlining similarity?*

5.1 Study Setup

5.1.1 Evaluation Dataset. We select the dataset constructed in Section 3 to evaluate CI-Detector. This dataset is constructed using 9 compilers, 4 optimizations, to 6 architectures, resulting in a total of 216 combinations. It contains several widely used projects, such as coreutils [11] and binutils [8]. 5,621,140 cross-inlining pairs are identified in this dataset.

5.1.2 Experiment Setting. CI-Detector is trained on cross-inlining pairs in the training dataset and predicts unknown pairs in the testing dataset. To generate the training and testing dataset, we split projects in *Dataset-Inlining* and *Dataset-NoInlining* by 80%, 10%, and 10% to generate the training projects, validation projects, and testing projects.

Though there may be cross-project reuse between the training projects and testing projects [39], we regard it consistent with the application scenario — the target binary always uses some existing code in one way or another.

In testing projects, we randomly generate 40,000 positive pairs and 40,000 negative pairs. These pairs are also used in Section 3.4 to evaluate existing binary code similarity detection works. We use the same set of testing pairs to facilitate comparison.

The binaries of the dataset are stripped to ensure that CI-Detector will not learn from symbols of the functions in cross-inlining pairs. We used the default parameters in GMN [57] to construct the GNN, where γ is set to 0.1 in Equation 1. CI-Detector is trained with 128 epochs to produce the cross-inlining models where each epoch consumes 40,000 positive pairs and 40,000 negative pairs randomly generated from training projects.

5.1.3 Evaluation Metrics. We use Accuracy, Precision, Recall, F1-score, and AUC to evaluate the effectiveness of CI-Detector. These metrics are widely used to evaluate the effectiveness of methods in classification tasks.

5.1.4 Implementation of CI-Detector. We use IDA Pro 7.3 [4], Capstone [9], and NetworkX [41] to construct ACFG for binary functions, and we use TensorFlow 1.14 [17] to implement GNN model. We refer to this repository [10] to help us with the implementation. The whole procedure is implemented in Python and we run all the experiments on a workstation equipped with Ubuntu 18.04, Intel Xeon Gold 6266C, and 1024GB DDR4 RAM.

5.2 RQ 1: Effectiveness of CI-Detector

In this section, we evaluate the effectiveness of CI-Detector by training it on the training projects and testing it on the testing pairs. As illustrated in Section 4.3, we select the threshold 0.55, which results in the best F1-score in the training pairs, to produce labels of testing pairs.

Table 2 shows the evaluation results of CI-Detector on cross-inlining testing pairs. As shown in Table 2, CI-Detector performs relatively well on all three cross-inlining patterns. In detail, CI-Detector can achieve an F1-score of 0.88 on *Leaf-Inlining* pairs, 0.89 on *Root-Inlining* pairs, and 0.88 on *Internal-Inlining* pairs. Especially, we noticed that CI-Detector achieves a relatively high recall in detecting cross-inlining pairs. For example, CI-Detector can detect 97% of *Leaf-Inlining* pairs, 99% of *Root-Inlining* pairs, and 96% of *Internal-Inlining* pairs. Meanwhile, CI-Detector can achieve an average accuracy of 81% in detecting these cross-inlining pairs.

We then respectively analyze the false negatives and false positives of CI-Detector. False negatives are the cases where the testing pair is positive while CI-Detector classifies it as negative. False positives are the cases where the testing pair is negative while CI-Detector classifies it as positive. As CI-Detector obtains a high recall, the false negatives rate is only 3%.

The false negatives are mainly due to the huge difference in function size of some positive cross-inlining pairs. One false negative example is the function *PUSH_CODE* in *Dataset-NoInlining* with the function *r_interpret* in *Dataset-Inlining*. Both functions are from the project gawk [12] compiled by gcc-5.5.0 with O3 to X86_64. We notice that this function pair has a huge difference in their function size. The function *PUSH_CODE* is only composed of 6 basic blocks with less than 30 instructions, while the function *r_interpret*

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

is composed of 857 basic blocks with 3,651 instructions. The huge difference not only makes it difficult to learn the cross-inlining relation between these two functions, but also makes it harder to find the location of the inlined function.

The false positives are mainly due to some small functions with a single functionality. For example, *free_token* is a function that frees the token sent to this function. However, many binary functions have the action of freeing some variables. They have similar actions with the function *free_token* but they do not inline such a function. This makes CI-Detector wrongly recognize such a function pair as a cross-inlining pair.

> **Answering RQ 1:** CI-Detector can effectively detect the cross-inlining binary code similarity. CI-Detector can achieve a precision of 81% and a recall of 97% when detecting cross-inlining pairs.

5.3 RQ 2: Compared with Existing Works

In this section, we compare the performance of CI-Detector with existing works. Apart from the works evaluated in Section 3.4, we also compare CI-Detector with Bingo [20] and Asm2Vec [30], which proposes strategies to simulate function inlining. Bingo and Asm2Vec use manual design rules to search for callee functions that should be inlined and then compare the binary functions after inlining to obtain the similarity. We use their released code and model for testing. Table 2 shows the performance of CI-Detector and existing works.

In general, CI-Detector exceeds all state-of-the-art works in terms of all metrics. Compared with existing works, CI-Detector obtained a 10%-30% improvement in precision, with a recall of more than 95%. Existing works often suffer from detecting *Leaf-Inlining* and *Internal-Inlining* pairs. However, CI-Detector can perform relatively well on all three inlining patterns.

The inlining strategies can help Bingo detect the *Root-Inlining* pairs with a recall of 95%, which exceeds all other existing works. However, its strategies only work for the *Root-Inlining* pairs. Its performance on *Leaf-Inlining* and *Internal-Inlining* pairs still suffers from low precision and recall. That is because the strategies of bingo can only help inline callee functions into query functions and target functions. When the query function is an inlined function, it cannot find the caller functions in the reverse direction. Besides, Asm2Vec does not perform well on the cross-inlining dataset. Even with the strategies of Bingo, it cannot accurately distinguish the cross-inlining pairs.

CI-Detector, which has respectively trained models for all cross-inlining patterns, can detect *Root-Inlining*, *Leaf-Inlining*, and *Internal-Inlining* pairs all with high coverage. Compared to the inlining strategies of Bingo, CI-Detector does not need the context of query functions and target functions. CI-Detector is more scalable for cross-inlining binary function similarity detection.

> **Answering RQ 2:** CI-Detector exceeds all state-of-the-art binary similarity detection works. Although Bingo has proposed strategies for *Root-Inlining* pairs, CI-Detector not only can detect the *Root-Inlining* pairs but also is effective in detecting *Leaf-Inlining* and *Internal-Inlining* pairs.

5.4 RQ 3: Contribution of Each Model

To evaluate the contribution of models in CI-Detector, we first create several variants of CI-Detector, including *Leaf-Inlining*, *Root-Inlining*, *Internal-Inlining*, and *Mixed* model. *Leaf-Inlining*, *Root-Inlining*, and *Internal-Inlining* model are methods that use the model trained only on one cross-inlining pattern. *Mixed* model is directly trained on all the cross-inlining patterns. Table 3 shows the evaluation results of them.

Table 3: Effectiveness of CI-Detectors and its variants

Method	Pattern	Accuracy	Precision	Recall	F1	AUC
Leaf	Leaf	0.86	0.80	0.95	0.87	0.87
	Root	0.88	0.82	0.98	0.89	0.93
	Internal	0.86	0.81	0.94	0.87	0.87
Root	Leaf	0.79	0.80	0.79	0.79	0.85
	Root	0.89	0.84	0.97	0.90	0.96
	Internal	0.82	0.82	0.83	0.82	0.87
Internal	Leaf	0.85	0.80	0.92	0.86	0.87
	Root	0.88	0.81	0.98	0.89	0.94
	Internal	0.86	0.81	0.94	0.87	0.88
Mixed	Leaf	0.84	0.80	0.89	0.85	0.86
	Root	0.87	0.81	0.96	0.88	0.92
	Internal	0.84	0.81	0.90	0.85	0.85
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

As shown in Table 3, single models including *Leaf-Inlining*, *Root-Inlining*, and *Internal-Inlining* models have interesting relations among them. For example, the *Root-Inlining* model performs extremely well on the *Root-Inlining* pairs but suffers a loss in recall of detecting *Leaf-Inlining* and *Internal-Inlining* pairs. That indicates that the *Root-Inlining* pattern cannot scale to the other two patterns. However, the *Leaf-Inlining* and *Internal-Inlining* models, which are

trained respectively on *Leaf-Inlining* and *Internal-Inlining* pairs, perform better on the *Root-Inlining* pairs than *Leaf-Inlining* and *Internal-Inlining* pairs. That indicates that the *Root-Inlining* pattern is easier to be learned compared with *Leaf-Inlining* and *Internal-Inlining* patterns.

The result of the *Mixed* model also indicates this relation. When training on all three cross-inlining patterns, we find that the *Mixed* model tends to perform better on the *Root-Inlining* pairs than the *Leaf-Inlining* and *Internal-Inlining* pairs. Especially, *Mixed* model suffer a loss in its recall of detecting all cross-inlining pairs, indicating that mixing these three inlining pattern makes the learning of cross-inlining pattern more difficult.

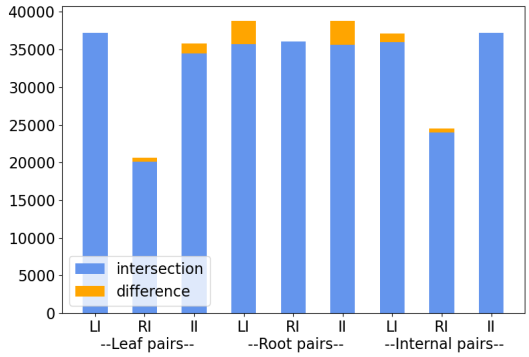


Figure 7: Relation of three cross-inlining models in retrieving positive pairs

CI-Detector improves the performance of detecting all three inlining patterns by combining all three cross-inlining models. Figure 7 presents the intersection and difference of the identified positive pairs detected by *Leaf-Inlining* (LI), *Root-Inlining* (RI), and *Internal-Inlining* (II) models. For example, the "RI" above the "–Leaf pairs" means the results of using *Root-Inlining* model to detect *Leaf-Inlining* pairs. The blue bar represents the intersection of positive pairs identified by *Root-Inlining* model and *Leaf-Inlining* model, and the orange bar represents the difference of *Root-Inlining* model to *Leaf-Inlining* model.

As shown in Figure 7, one model can help another model recover hidden positive pairs. For example, *Leaf-Inlining* and *Internal-Inlining* models help *Root-Inlining* model recover thousands of positive *Root-Inlining* pairs. Besides, *Internal-Inlining* model can help *Leaf-Inlining* model recover *Leaf-Inlining* positive pairs, and *Leaf-Inlining* model can help *Internal-Inlining* model recover *Internal-Inlining* positive pairs. As a result, CI-Detector can achieve a recall of 97% on detecting *Leaf-Inlining* pairs, 99% on detecting *Root-Inlining* pairs, and 96% on detecting *Internal-Inlining* pairs.

Answering RQ 3: The *Leaf-Inlining*, *Root-Inlining*, and *Internal-Inlining* models can help each other to recover more positive cross-inlining pairs, which helps CI-Detector to achieve high recall when detecting all three kinds of cross-inlining pairs.

5.5 RQ 4: Efficiency of CI-Detector

In this section, we will evaluate the efficiency of CI-Detector. We will first present the training time of CI-Detector. Then we compare the testing time of CI-Detector with existing works.

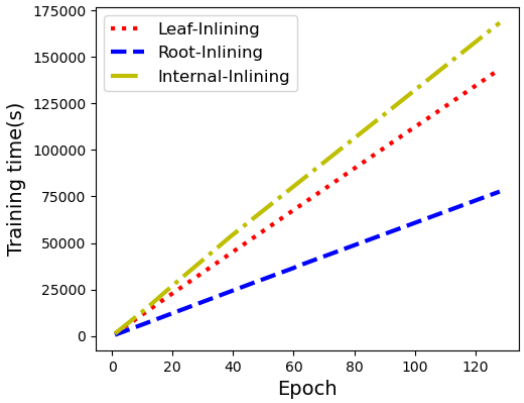


Figure 8: Training time of CI-Detector

As CI-Detector contains three models training respectively for three cross-inlining patterns. We record the training time of these three cross-inlining models as shown in Figure 8. In general, training on *Root-Inlining* pairs in 128 epochs costs about 20 hours, training on *Leaf-Inlining* pairs costs about 37 hours, and training on *Internal-Inlining* pairs costs about 48 hours.

Considering that CI-Detector needs to be trained only once, the training time is acceptable.

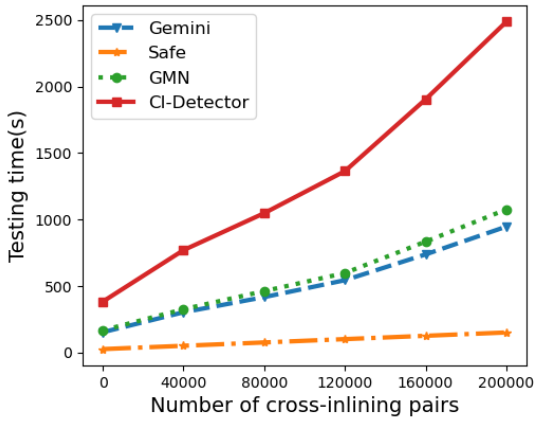


Figure 9: Testing time of CI-Detector

Figure 9 shows the testing time of Gemini, Safe, GMN, and CI-Detector. Bingo and Asm2vec need to perform the program-level comparison to calculate the similarity between two functions, thus we exclude them. As shown in Figure 9, Safe spends less than 0.001s to process a testing pair as it only uses a self-attentive neural network. The GNN-based methods including Gemini and GMN spend 0.005s to process a testing pair on average. CI-Detector, which uses three GNN models, spends about 0.013s to process a testing pair.

Though CI-Detector spends more time than existing works, it is still efficient as it only needs about 2/3 hours to process 200,000 testing pairs.

Answering RQ 4: CI-Detector can efficiently detect the cross-inlining similarity. On average, the models in CI-Detector only need 0.013s to produce the similarities of a cross-inlining pair.

6 THREATS TO VALIDITY

In this section, we will discuss the threats to validity.

Threats to internal validity. The threat to internal validity is that we use IDA Pro to construct the ACFGs of binary functions. As pointed out by study [69], accurate disassembling and binary analysis is not easy. The errors in disassembling may affect the ACFG construction and further affect the results.

Threats to external validity. The threat to external validity is the diversity of compilation settings. In this paper, though we have compiled the project with 9 compilers, and 4 optimizations, to 6 architectures, there may be other compilation settings such as non-default optimizations [73]. The inlining patterns learned by our model may not scale to projects with different inlining conventions. However, our method can be easily retrained using other datasets, as long as these datasets can be labeled with inlining labels.

Threats to construct validity. The threat to the construct validity is that our labeling method relies on some third-party tools such as IDA Pro and Dwarf debug tools. As also mentioned in [49], these tools can be inaccurate, thus further affecting the identification of function inlining. To help mitigate the influence, we have investigated enormous tools and chosen the most reliable ones. For example, IDA Pro is a state-of-the-art commercial tool and has been widely used in academia [32, 69] and industry [1, 2].

7 CONCLUSION

In this paper, we propose a pattern-based model named CI-Detector for cross-inlining matching. CI-Detector uses the ACFG to represent the semantics of binary functions and GNN to embed binary functions into vectors. We summarize three cross-inlining patterns in the cross-inlining dataset and CI-Detector respectively trains a model for these three cross-inlining patterns. The testing pairs are input to these three models and all the produced similarities are aggregated to produce the final similarity. We conduct several experiments to evaluate CI-Detector. Results show that CI-Detector can detect cross-inlining pairs with a precision of 81% and a recall of 97%, which exceeds all state-of-the-art works.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (2022YFB2703500), National Natural Science Foundation of China (62232014, 62272377, 62293501, 62293502, 72241433, 61721002, 62032010, 62002280, 62372368, 62372367), CCF-AFSG Research Fund, China Postdoctoral Science Foundation (2020M683507, 2019TQ0251, 2020M673439), and Young Talent Fund of Association for Science and Technology in Shaanxi, China.

REFERENCES

[1] 2020. Reveiws 1 - SciTools. https://news.sophos.com/en-us/2020/04/26/asnarok/. [Online; accessed 3-September-2021].
[2] 2020. What's up, Emotet? CERT Polska. https://cert.pl/en/posts/2020/02/whats-up-emotet/. [Online; accessed 3-September-2021].
[3] 2021. Gnulib - The GNU Portability Library. https://www.gnu.org/software/gnulib/. [Online; accessed 23-April-2022].
[4] 2021. IDA Pro Disassembler and Debugger - Hex Rays. https://www.hex-rays.com/ida-pro/. [Online; accessed 20-April-2022].
[5] 2022. Inline expansion. https://en.wikipedia.org/wiki/Inline_expansion. [Online; accessed 23-April-2022].
[6] 2022. Synopsys 2022 open source security and risk analysis report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html. [Online; accessed 29-May-2023].
[7] 2022. VDC Research White Paper | Finding Sources of Security in the Complex Software Supply Chains of Tomorrow. https://codesonar.grammatech.com/wp-form-vdc-research-software-supply-chain. [Online; accessed 29-May-2023].
[8] 2023. Binutils - GNU Porject. https://www.gnu.org/software/binutils/. [Online; accessed 25-July-2023].
[9] 2023. capstone-PyPI. https://pypi.org/project/capstone/. [Online; accessed 13-July-2022].
[10] 2023. Cisco-Talos/binary-function-similarity. https://github.com/Cisco-Talos/binary_function_similarity. [Online; accessed 13-July-2022].
[11] 2023. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/. [Online; accessed 25-July-2023].
[12] 2023. Gawk - GNU Project. https://www.gnu.org/software/gawk/. [Online; accessed 25-July-2023].
[13] 2023. NVD - CVE-2014-3569. https://nvd.nist.gov/vuln/detail/cve-2014-3569. [Online; accessed 6-June-2023].
[14] 2023. NVD - CVE-2015-1792. https://nvd.nist.gov/vuln/detail/CVE-2015-1792. [Online; accessed 29-May-2023].
[15] 2023. OpenSSL. https://www.openssl.org/.
[16] 2023. Source code and dataset. . https://github.com/island255/cross-inlining_binary_function_similarity. [Online; accessed 21-July-2023].
[17] 2023. tensorflow-PyPI. https://pypi.org/project/tensorflow/. [Online; accessed 13-July-2022].
[18] Pär Andersson. 2009. Evaluation of inlining heuristics in industrial strength compilers for embedded systems.
[19] 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, 1 (2021), 1–21.
[20] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.
[21] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. 2008. An Adaptive Strategy for Inline Substitution. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4959)*. Springer, 69–84. https://doi.org/10.1007/978-3-540-78791-4_5
[22] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*. PMLR, 2702–2711.
[23] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *Acm sigplan notices* 51, 6 (2016), 266–280.
[24] 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.
[25] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 392–404. https://doi.org/10.1145/3173162.3177157
[26] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.
[27] Jack W. Davidson and Anne M. Holler. 1992. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Trans. Software Eng.* 18, 2 (1992), 89–102. https://doi.org/10.1109/32.121752
[28] Jeffrey Dean and Craig Chambers. 1994. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*. ACM, 273–282. https://doi.org/10.1145/182409.182489
[29] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2016. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 461–470.
[30] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code

obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[31] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2169–2185. https://doi.org/10.1145/3133956.3134048

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

[33] Irène A Durand and Robert Strandh. 2018. Partial Inlining Using Local Graph Rewriting. In *11th European Lisp Symposium*.

[34] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[35] Muyue Feng, Weixuan Mao, Zimu Yuan, Yang Xiao, Gu Ban, Wei Wang, Shiyang Wang, Qian Tang, Jiahuan Xu, He Su, Binghong Liu, and Wei Huo. 2019. Open-Source License Violations of Binary Software at Large Scale. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 564–568. https://doi.org/10.1109/SANER.2019.8667977

[36] Muyue Feng, Zimu Yuan, Feng Li, Gu Ban, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xue, and Wei Huo. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1038–1049. https://doi.org/10.1109/ASE.2019.00100

[37] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.

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

[39] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 291–301.

[40] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.

[41] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

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

[43] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*. ACM, 63–72. https://doi.org/10.1145/1985441.1985453

[44] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-Architecture Binary Semantics Understanding via Similar Code Comparison. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 57–67. https://doi.org/10.1109/SANER.2016.50

[45] Jan Hubicka. 2004. The GCC call graph module: a framework for inter-procedural optimization.

[46] Wen-mei W. Hwu and Pohua P. Chang. 1989. Inline Function Expansion for Compiling C Programs. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. ACM, 246–257. https://doi.org/10.1145/73141.74840

[47] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 81–96. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang

[48] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. [n. d.]. 1-to-1 or 1-to-n? Investigating the effect of function inlining on binary similarity analysis. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).

[49] 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 Trans. Softw. Eng. Methodol.* 32, 4, Article 87 (may 2023), 26 pages. https://doi.org/10.1145/3561385

[50] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 2022. Comparing One with Many–Solving Binary2source Function Matching Under Function Inlining. *arXiv preprint arXiv:2210.15159* (2022).

[51] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[52] Ulf Kargén and Nahid Shahmehri. 2017. Towards robust instruction-level trace alignment of binary code. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 342–352. https://doi.org/10.1109/ASE.2017.8115647

[53] Dongjin Kim, Seong-je Cho, Sangchul Han, Minkyu Park, and Ilsun You. 2014. Open Source Software Detection using Function-level Static Software Birthmark. *J. Internet Serv. Inf. Secur.* 4, 4 (2014), 25–37. https://doi.org/10.22667/JISIS.2014.11.31.025

[54] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. (2020). arXiv:2011.10749 [cs.SE]

[55] TaeGuen Kim, Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2019. Binary executable file similarity calculation using function matching. *J. Supercomput.* 75, 2 (2019), 607–622. https://doi.org/10.1007/s11227-016-1941-2

[56] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[57] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 3835–3845. http://proceedings.mlr.press/v97/li19d.html

[58] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130* (2017).

[59] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of malicious code: insights into the malicious software industry. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, Robert H'obbes' Zakon (Ed.). ACM, 349–358. https://doi.org/10.1145/2420950.2421001

[60] 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.

[61] Xin Liu, Yixiong Wu, Qingchen Yu, Shangru Song, Yue Liu, Qingguo Zhou, and Jianwei Zhuge. 2022. PG-VulNet: Detect Supply Chain Vulnerabilities in IoT Devices using Pseudo-code and Graphs. In *ESEM '22: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki Finland, September 19 - 23, 2022*, Fernanda Madeiral, Casper Lassenius, Tayana Conte, and Tomi Männistö (Eds.). ACM, 205–215. https://doi.org/10.1145/3544902.3546240

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

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

[64] Andrew McCallum, Kamal Nigam, et al. 1998. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, Vol. 752. Madison, WI, 41–48.

[65] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7839)*, Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.). Springer, 92–109. https://doi.org/10.1007/978-3-642-37682-5_8

[66] 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 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 253–270. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming

[67] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015,*

Hamburg, Germany, May 26-28, 2015, Proceedings (IFIP Advances in Information and Communication Technology, Vol. 455), Hannes Federrath and Dieter Gollmann (Eds.). Springer, 416–430. https://doi.org/10.1007/978-3-319-18467-8_28

[68] Dhaval Miyani, Zhen Huang, and David Lie. 2017. BinPro: A Tool for Binary Source Code Provenance. *CoRR* abs/1711.00830 (2017). arXiv:1711.00830 http://arxiv.org/abs/1711.00830

[69] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 833–851.

[70] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[71] Ashkan Rahimian, Philippe Charland, Stere Preda, and Mourad Debbabi. 2012. RESource: A Framework for Online Matching of Assembly with Open Source Code. In *Foundations and Practice of Security - 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7743)*. Springer, 211–226. https://doi.org/10.1007/978-3-642-37119-6_14

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

[73] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. *Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035

[74] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. 2021. Osprey: A fast and accurate patch presence test framework for binaries. *Comput. Commun.* 173 (2021), 95–106. https://doi.org/10.1016/j.comcom.2021.03.011

[75] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 319–330. https://doi.org/10.1109/ASE.2017.8115645

[76] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[77] 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 *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 873–884. https://doi.org/10.1109/ICSE43902.2021.00084

[78] 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*. 376–387.

[79] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149.

[80] 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.

[81] 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 *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[82] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 887–902.

[83] Peng Zhao and José Nelson Amaral. 2003. To Inline or Not to Inline? Enhanced Inlining Decisions. In *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 2958)*. Springer, 405–419. https://doi.org/10.1007/978-3-540-24644-2_26

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