1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis

ANG JIA, MING FAN, WUXIA JIN, XI XU, and ZHAOHUI ZHOU, Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi’an Jiaotong University

QIYI TANG, SEN NIE, and SHI WU, Tencent Security Keen Lab

TING LIU, Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Cyber Science and Engineering, Xi’an Jiaotong University

Binary similarity analysis is critical to many code-reuse-related issues, where function matching is its fundamental task. "1-to-1" mechanism has been applied in most binary similarity analysis works, in which one function in a binary file is matched against one function in a source file or binary file. However, we discover that the function mapping is a more complex problem of "1-to-n" (one binary function matches multiple source functions or binary functions) or even "n-to-n" (multiple binary functions match multiple binary functions) due to the existence of function inlining, different from traditional understanding.

In this article, we investigate the effect of function inlining on binary similarity analysis. We carry out three studies to investigate the extent of function inlining, the performance of existing works under function inlining, and the effectiveness of existing inlining-simulation strategies. Firstly, a scalable and lightweight identification method is designed to recover function inlining in binaries. 88 projects (compiled in 288 versions and resulting in 32,460,156 binary functions) are collected and analyzed to construct four inlining-oriented datasets for four security tasks in the software supply chain, including code search, OSS (Open Source Software) reuse detection, vulnerability detection, and patch presence test. Datasets reveal that the proportion of function inlining ranges from 30–40% when using O3 and sometimes can reach nearly 70%. Then, we evaluate four existing works on our dataset. Results show most existing works neglect inlining and use the "1-to-1" mechanism. The mismatches cause a 30% loss in performance during code search and a 40% loss during vulnerability detection. Moreover, most inlined functions would be ignored during OSS reuse detection and patch presence test, thus leaving these functions risky. Finally, we analyze two inlining-simulation strategies on our dataset. It is shown that they miss nearly 40% of the inlined functions, and there is still a large space for promotion. By precisely recovering when function inlining happens, we discover that inlining is usually cumulative when optimization increases. Thus, conditional inlining and incremental inlining are recommended to design a low-cost and high-coverage inlining-simulation strategy.

This work was supported by National Natural Science Foundation of China (61902306, 62002280, 61721002, 61833015), and sponsored by the CCF-Tencent Open Research Fund, the Fundamental Research Funds for the Central Universities (xxj022019001, xzy012020009), China Postdoctoral Science Foundation (2020M683507, 2019TQ0251, 2020M673439), and Youth Talent Support Plan of Xi’an Association for Science and Technology (095920201303).

Authors’ addresses: A. Jia, M. Fan (corresponding author), W. Jin, X. Xu, Z. Zhou, and T. Liu (corresponding author), Xi’an Jiaotong University, Shaanxi 710049, China; emails: jiaang@stu.xjtu.edu.cn, {mingfan, jinwuxia}@mail.xjtu.edu.cn, {xx19960325, zhzhou}@stu.xjtu.edu.cn, tingliu@mail.xjtu.edu.cn; Q. Tang, S. Nie, and S. Wu, Tencent Security Keen Lab, Shanghai 200233, China; emails: {dodgetang, snie, shiwu}@tencent.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Association for Computing Machinery.
1049-331X/2023/05-ART87 $15.00
https://doi.org/10.1145/3561385

ACM Transactions on Software Engineering and Methodology, Vol. 32, No. 4, Article 87. Pub. date: May 2023.
1 INTRODUCTION

According to a report published by Gartner [37], over 90% of the development organizations stated that they rely on open source components. Although proper use of open-source code helps reduce the development cost, improper use of open-source code may bring legal and security risks. Developers may unintentionally violate open-source licenses, consequently causing great financial loss to software companies. For example, Cisco and VMware were exposed to serious legal issues because they did not adhere to the licensing terms of the Linux kernel [1, 2]. Moreover, vulnerabilities are easy to exploit in open-source software (OSS) since its code is accessible. Software systems that use the vulnerable code and do not patch it in time will suffer from security risks [67]. Improper dependence on third-party software in the software supply chain makes the situation worse.

Binary code similarity analysis works are proposed to resolve these code-reuse-related issues. The fundamental task of binary code similarity analysis is to match the query binary function and target source/binary functions. Here, the query function is the function in a binary absence of its origin, and the target function represents functions that may be related to a license, vulnerability, or patch. Many binary code similarity analysis applications [20, 23, 34, 35, 39, 45, 51, 66, 68], including code search [25, 26, 46, 62, 66], OSS reuse detection [20, 27, 33, 34, 38, 41, 47, 54], vulnerability detection [24, 36] and patch presence test [64, 67] are always using the “1-to-1” matching mechanism, i.e., one function in a binary file is matched against one function in a source file or binary file. However, we discover that such mapping is usually a more complex problem of “1-to-n” or “n-to-n” due to the existence of function inlining [10], which differs from traditional assumptions.

Figure 1 shows a motivating example when searching a 1-day vulnerability under inlining. Figure 1(a) shows the source code of vulnerable function scm_send in the project Linux for CVE-2020-16593. Considering that code reuses of this vulnerable function may exist, the vulnerability would be propagated to disparate binaries. As the 1-day vulnerability is introduced by code reuse, existing vulnerability detection works usually assume that functions with the same 1-day vulnerability usually have similar contents. Thus, they detect the influenced binaries by searching whether there is a function in the binary similar to the vulnerable function in the source code. To support this process, Figure 1(c) depicts the complete Control Flow Graph (CFG) of a binary function netlink_sendmsg that will be checked, with a partial CFG shown in Figure 1(b). Each node represents a basic block, and the array represents the control flow between them.

Following the “1-to-1” mechanism, we cannot find an individual binary function that is matching to the function scm_send. Analyzing the debug information, we found the scm_send’s code is compiled into another function named netlink_sendmsg. However, the existing works calculate a low similarity score between the function scm_send and the target binary function.
Function inlining causes the failure of the “1-to-1” matching mechanism. Figure 1(d) shows inlining happens to function netlink_sendmsg. Function scm_send, called by netlink_sendmsg, is inlined into netlink_sendmsg. When a function inlining occurs, it replaces the call-site statement inside a caller with the function body of the callee. As a result, the produced binary function will contain multiple source functions. Even more, the function inlining can be nested, meaning that an inlined function may further inline another function. As shown in Figure 1(d), the scm_send, inlined by netlink_sendmsg, inlines scm_set_cred. We can observe that the mapping between binary function netlink_sendmsg and its original source functions is no longer a “1-to-1” mapping but a “1-to-n” issue — one binary function maps multiple source functions, sometimes even in a nested inlining manner. Thus, the existing “1-to-1” matching mechanism, which tries to match two functions with unequal semantics, suffered the expected failure.

“1-to-n” issues bring a great challenge to existing binary code similarity analysis works. Caused by the function inlining, the binary2binary function mapping in matching has become “1-to-n” and even “n-to-n”. However, to the best of our knowledge, few existing works have systematically studied the effect of function inlining on binary code similarity analysis works. It is still unclear the extent of function inlining and its impact on existing binary similarity analysis works. This article will address this issue, and concretely we will investigate the following three research questions (RQs).

RQ1: To what extent will inlining happen during compilation? To answer this RQ, we will evaluate the frequency and degree of function inlining, which can demonstrate the wild presence of function inlining in binary code similarity analysis tasks. This RQ will be answered in Section 4.

RQ2: What is the effect of function inlining on the performance of existing binary code similarity analysis works? To answer this RQ, we will evaluate the performance of existing works that do not consider function inlining or do not fully handle function inlining (which account for the majority). The decrease in performance indicates the importance of considering function inlining. We will answer this RQ in Section 5.

RQ3: Can existing inlining-simulation strategies solve them? Existing works have proposed strategies to simulate the results of function inlining (here, we call them inlining-simulation strategies). To answer this RQ, we will evaluate the effectiveness of existing inlining-simulation strategies.
strategies from the inlining cost and coverage of inlined functions. The shortcomings of existing inlining-simulation strategies indicate directions for improvement. This RQ will be answered in Section 6.

We first construct four inlining-oriented datasets for four security tasks in the software supply chain, including code search, OSS reuse detection, vulnerability detection, and patch presence test to support our study. Furthermore, we propose a scalable and lightweight identification method that can automatically recover function inlining in binaries. Based on our labeled dataset, we then investigate the extent of function inlining, the impact of function inlining on existing binary code similarity works, and the effectiveness of existing inlining-simulation strategies. Finally, we discuss the interesting findings and give our suggestions for designing more effective inlining-simulation strategies.

Our main contributions are listed below:

— To the best of our knowledge, we are the first to comprehensively evaluate the effect of function inlining on four security tasks in the software supply chain, including code search, OSS reuse detection, vulnerability detection, and patch presence test.
— We create the first function inlining dataset by designing a scalable and lightweight identification method that can automatically recover function inlining in binaries (Section 3).
— Our study recovers the high frequency and high degree of function inlining (Section 4). Averagely the proportion of function inlining can range from 40% to 70%, and a binary function can inline 2–4 source functions.
— We conduct five experiments to evaluate the performance of existing binary code similarity works under function inlining (Section 5). Code search suffers a 30% decline in detecting functions with inlining while OSS reuse detection and patch presence test miss most inlined functions.
— We take a deep analysis of existing inlining-simulation strategies (Section 6) and point out the directions for designing more effective strategies (Section 7).

To facilitate further research, we have made the source code\(^3\) and dataset\(^4\) publicly available.

2 PRELIMINARY

This section will introduce the four security tasks in the software supply chain and the research scope of our work.

2.1 Software Supply Chain Security Tasks

Most software today is not developed entirely from scratch. Instead, developers rely on a range of third-party resources to create their applications. On one hand, using pre-built libraries helps to differentiate their software, finish projects quicker, reduce costs, and stay competitive. On the other hand, dependence on risky third-party libraries brings software supply chain security risks. To ensure software supply chain security, one should have a clear view of and control over these three-party libraries. Code search\([22, 26, 65, 66]\) aims at finding functionally identical functions, which further help identify whether there are dependencies on three-party libraries. Moreover, OSS reuse detection\([20, 27, 33, 34, 38, 41, 47, 54, 66]\) can help find reused OSS to avoid an illegal use of licensed open-sourced code, and vulnerability/patch detection\([22, 24, 36, 64, 67]\) help examine whether there are vulnerabilities laying in the third-party libraries.

\(^3\)https://github.com/island255/TOSEM2022.
\(^4\)https://doi.org/10.5281/zenodo.6675280.
In this section, we will introduce these four security-related tasks and their representative works.

2.1.1 Code Search. As defined in many existing works [22, 26, 65, 66], code search aims at finding functionally identical functions. In traditional experiment settings, existing binary2binary code search works set the target as matching the binary functions with the same function name but compiled with different compilation settings, while binary2source code search works match these binary functions to the source function with the same name. A ranking list of similar functions will be returned as the result of a query function. SAFE [46] is a representative work in binary2binary code search, which converts functions into embeddings and searches functions using embedding similarities.

2.1.2 OSS Reuse Detection. OSS reuse detection [20, 27, 33, 34, 38, 41, 47, 54, 66] aims at finding the reuse of source code in compiled binaries. Here, binaries are compared with open source repositories to recover the OSS reuse in binaries. Currently, the state-of-the-art OSS reuse detection tool is CodeCMR [66], which is a function-level binary2source detection tool and can achieve 90% recall on retrieving the source function that a binary function is compiled from.

2.1.3 Vulnerability Detection. Code similarity based vulnerability detection [22, 24, 36] aim at detecting 1-day vulnerabilities by comparing target functions with the vulnerable function. Usually, in existing works, 1-day vulnerabilities are assumed to have similar contexts as they usually spread to different projects due to the reuse of common software components, such as OpenSSL [17]. Vulseeker [36] is a representative work that searches vulnerable functions by comparing target functions with vulnerable functions.

2.1.4 Patch Presence Test. As vulnerable functions and patched functions only have little difference when the patch is small, existing vulnerability detection works may confuse them. Patch presence test [64, 67] tries to further differentiate them and identify whether a vulnerable function has been patched. Currently, a representative work, BinXray [64], leverages the difference between vulnerable functions and patched functions to find whether the target function is more similar to the vulnerable function or the patched function. The difference depends on how they identify the range of vulnerabilities and patches.

Code search, OSS reuse detection, vulnerability detection, and patch presence test are four important security tasks in the software supply chain, but currently, none of the existing works have fully considered the problem of function inlining. In this article, we will focus on this question and reveal the challenges that inlining has brought to these applications.

2.2 Research Scope

Before introducing our work, we first clarify our research scope. Our work investigates the impact of inlining strategies used in common compilers on static function-level binary code similarity analysis works. Other function inlining strategies or instruction-level, basic-block-level, and file-level methods are beyond our scope.

We focus on C/C++ programs compiled by GCC and Clang with O0-O3. The main architecture of binaries is X86-64, but we also analyze datasets compiled in eight architectures. Moreover, we only consider the inlining of user-defined (UD) functions in this article. The inlining of library functions is similar to UD functions, but library functions may differ when the compilation environment changes.
3 DATASET CONSTRUCTION

A few studies have researched the impact of function inlining on binary code similarity analysis work. An inlining-oriented dataset is needed. To fill the gap, we will first collect 88 projects and propose a scalable and lightweight identification method to recover function inlining in binaries automatically. As illustrated in Figure 2, we first collect source projects and compile them into binaries under diverse configurations (Section 3.1). Based on the source code and binaries, we propose an automatic identification method to identify function inlining in binaries (Section 3.2).

3.1 Dataset Collection

In this work, we focus on C/C++ programs and two popular compilers, GCC [11] and Clang [7]. We have collected four datasets by employing and extending Binkit [42], a binary code similarity analysis benchmark. The compositions and compilation settings of these datasets are listed in Table 1.

**Dataset I—for code search.** Dataset I was composed of the 51 packages from GNU projects, compiled by 9 compilers, 4 optimizations, to 8 architectures (shown in Table 2), resulting in 288 compilation versions containing 67,680 binaries and 18,783,986 functions. Dataset I was collected from Binkit [42]. This dataset contains many widely-used packages such as Coreutils [8], which have been extensively used in binary similarity detection works [22, 29, 44, 61, 63, 66].

**Dataset II—for OSS reuse detection.** Dataset II was extended from a partial reuse dataset used in ISRD [63], containing 24 programs with 74 partial reuses. It was originally compiled by GCC with O2 in X86-64. We extended it to four optimizations by using gcc-8.2.0. Dataset II was not extended to other architectures because existing OSS reuse detection works did not include them. Dataset II was composed of 96,085 source functions and 190,611 binary functions.

**Dataset III—for vulnerability/patch detection.** Dataset III was constructed from the dataset used in BinXray [64], containing 479 CVEs and 6238 vulnerable functions. We generated two versions of Dataset III, including Dataset III (without additional compilation flags) and Dataset III-NI.

---

**Table 1. Composition of Four Datasets**

| Dataset | # of Projects | Representative Projects | Compilation Setting | Versions | Binary Functions |
|---------|---------------|-------------------------|---------------------|----------|------------------|
| I       | 51            | Coreutils [8], Findutils [9] | Table 2            | 288      | 18,783,986       |
| II      | 24            | bzip2 [14], minizip [16] | gcc-8.2.0, X86-64, O0-O3 | 4        | 190,611         |
| III     | 12            | OpenSSL [17], Binutils [6] | gcc-8.2.0, X86-64, O0-O3, inline, noinline | 8        | 8,635,362       |
| IV      | 1             | Chromium [12]             | clang, debug, release | 2        | 4,850,197        |

**Table 2. Compilers, Optimizations, and Architectures used in Dataset I**

| Compilers | gcc-4.9.4, gcc-5.5.0, gcc-6.4.0, gcc-7.3.0, gcc-8.2.0, clang-4.0, clang-5.0, clang-6.0, clang-7.0 |
| Optimizations | O0, O1, O2, O3 |
| Architectures | X86-32, X86-64, ARM-32, ARM-64, MIPS-32, MIPS-64, MIPSeb-32, MIPSeb-64 |

---

5https://www.gnu.org.
Investigating the Effect of Function Inlining on Binary Similarity Analysis

Fig. 3. Example of extending line-level mapping to function-level mapping.

(with flag “-fnoinline” to forbid inlining) to facilitate the evaluation of patch detection without cross optimizations. Dataset III was composed of 8,635,362 binary functions.

**Dataset IV—For application binaries.** Apart from these three datasets, we also constructed a dataset by compiling chromium [12] with Clang in its debug and release optimization to X86-64 to form the dataset IV. Dataset IV was comprised of 4,850,197 binary functions. This dataset provides a vision of function inlining in large application binaries.

### 3.2 Function Inlining Identification

In this section, we will first illustrate terminologies that we will use throughout our article and then introduce our function inlining identification method.

#### 3.2.1 Terminology.

To make the representation clear, we first illustrate our notions of the functions from both source and binary. Usually, as shown in Figure 1(d), when function inlining happens, a binary function (netlink_sendmsg) is compiled from the original source function (netlink_sendmsg) and some inlined source functions (scm_send, scm_set_cred and etc.). Here, we named the produced binary function (netlink_sendmsg) as **BFI** (binary function with inlining), the original source function (netlink_sendmsg) as OSF, and the inlined source functions (scm_send, scm_set_cred, and etc.) as **ISFs**.

Moreover, the binary function is BF and the source function is SF. These notions will be used throughout the article.

#### 3.2.2 Function Inlining Identification Method.

To identify function inlining, we first construct function mappings from source code to binaries, which can also serve as the ground truth for binary code similarity analysis. When compiling a program in debug mode, the compiler will record the mapping from binary instructions to source instructions. Following that mapping, we further find the binary function that binary instructions belong to and the source function that source instructions belong to. Thus, function-to-function mapping can be generated.

The function mappings identification method leverages the line mapping implied by the relation between binary address and source code line, provided in the debug mode. By extending line mapping results, the method will generate a function-level mapping between binary functions and source functions. We will introduce the algorithm with an example in Figure 3.

Our method first compiles source code with the “-g” option and leverages Dwarf [6] to produce the .debug_line section in binary. To extract the line-level mapping, we use Readelf [7] to parse binaries. As shown in Figure 3, the line mapping contains the mapping between source file lines and binary addresses.

---

6 [http://dwarfstd.org/Dwarf5Std.php](http://dwarfstd.org/Dwarf5Std.php).
7 [https://man7.org/linux/man-pages/man1/readelf.1.html](https://man7.org/linux/man-pages/man1/readelf.1.html).
Next, the method extracts line-to-function mapping in the source code by employing Understand\(^8\) and extracts address-to-function mapping in the binary code by employing IDA Pro.\(^9\) Understand and IDA Pro are two state-of-the-art commercial tools and have been widely used in academia \([28, 49, 52]\) and industry \([3–5]\). Understand is a static analysis tool to identify source entities (e.g., files, functions) and dependencies. By exploring source entities, we can infer which entity the code line belongs to. IDA Pro recovers the function boundary \([19]\) from binaries, outputting binary address to binary function. For example, in Figure 3, source line 287 belongs to the source function usage, and binary address 0x401C81 belongs to binary function usage.

Finally, our method extends the above line-level mapping results to function-level mapping by combining these three mappings. As shown in Figure 3, now we can get the function mapping between source function usage and binary function usage.

Our datasets collected in Section 3.1 are labeled by our automatic method. According to the mapping results between BFs and SFs, we can easily identify BFIs produced by function inlining (i.e., BFI). Simply, we identify BFIs mapped to more than one SFs as BFIs and those mapped to one SF as NBFs. By identifying the OSF or NSF from which BFIs are compiled, we can also easily establish binary2binary function-level mappings. Our datasets with function-level mappings can support the study in our work.

Besides, our method does not rely on any modification of the compilers and can be applied to any binaries compiled in debug mode. Note that Understand and IDA Pro can be substituted with other tools with similar functionalities. We also implement a tool using tree-sitter\(^10\) and Ghidra,\(^11\) facilitating free usage.

### 3.2.3 Effectiveness of Function Inlining Identification Method

In the ground truth construction procedure of most binary code similarity analysis works \([22, 62, 63]\), binary function pairs that need to be matched are identified through the same source lines they mapped. Similar to them, our function inlining identification is extended to identify the source functions that the binary function mapped. Theoretically, our function inlining identification conforms to established conventions and is precise in identifying inlined functions.

However, the effectiveness of our method may still be influenced by some implementation issues. To evaluate the effectiveness of the function inlining identification method, we first need to create a dataset with explicit ground truths. To produce explicit information about whether a function is inlined, we add some arguments to GCC and Clang and recompile the dataset I to produce the ground truth.

When using GCC to compile projects, we add a flag `-fdump-ipa-inline` to let the compiler produce a file with the suffix “.inline” recording the inlining decisions of GCC. However, this log can only be produced for a single source file. Thus it cannot be used for the inlining identification of our full dataset but can only serve as the ground truth to evaluate our function inlining identification method.

When using Clang to compile projects, we use the flag `-fllvm-opt-temps` to let the LLVM \([13]\) produce LLVM IR with debug information. The debug information contains the line mapping and the source function that a line belongs to. However, compiling projects with this flag does not always succeed. We create a dataset by compiling Coreutils with default four Ox optimizations.

\(^8\)https://www.scitools.com/.
\(^9\)https://www.hex-rays.com/ida-pro/.
\(^10\)https://github.com/tree-sitter/tree-sitter.
\(^11\)https://github.com/NationalSecurityAgency/ghidra.
After constructing the dataset, we run our function inlining identification method on it and manually check whether our results are complete and accurate. Considering the manual effort, we respectively randomly select 1,000 functions from the GCC/Clang dataset and examine their correctness. Here are the results:

In general, our method obtained a 100% precision in getting the ISFs and an 88% recall in recovering the source function that source lines belong to. In all 2,000 functions, all the recovered source functions of the binary function are accurate, but not all the source functions are covered. 545 binary functions remain partially covered, and about 12% (23,409/193,583) of line-to-address mappings cannot find the source function that the source line belongs to. This issue widely exists in the many static source parsers, including Understand and tree-sitter. To cover this gap, we make additional efforts by combining debug information and manual inspections.

For example, in Coreutils v8.29, we found that 2,544 lines in 117 files remain unresolved by Understand. Leveraging the debug information in LLVM IR produced by Clang and LLVM, we can reduce the unresolved lines from 2,544 to 597 and unresolved files from 117 to 31. Of the remaining 31 files, 16 files are project files, and 15 files are other system library files. Our datasets do not consider system library functions. Finally, we manually analyzed the 16 unresolved project files.

We have added our manual effort to aid the identification of source functions. We only analyze these fully recovered binary functions in our subsequent studies to ensure precise study results.

3.3 Application Matching Labels Considering Function Inlining

In traditional binary code similarity analysis works, without considering function inlining, labels are only attached to the functions with the same name. However, when considering function inlining, the composition of binary function becomes much more complicated. A thorough procedure of how we attach labels to functions needs to be introduced. In detail, as these four security tasks have different objectives, we separately discuss the labeling procedures in them. Here are the details.

**Code search.** To illustrate labeling in code search, we first define the query function and target functions. Usually, the query function is a BF unknown of its origin, and the target functions are known BFs or SFs. Here, for binary2binary code search, if the query BF and the target BF origin from the same OSF/NSF, we label this function pair as a matching pair. Similarly, if the query BF originated from the target OSF/NSF, we label this function pair as a matching pair. However, as code search aims at finding functionally identical functions, we will not mark the query BF and the target ISF as a matching pair.

**OSS reuse detection.** To construct ground truth in OSS reuse detection, we first detect similarity between source functions of projects by using JPlag [53] following ISRD [63]. Then, we identify partial reuse by selecting function pairs whose similarity is more than 90%. Next, we map the source functions of the query project to its binary functions. This procedure recorded cases where the SFs serve as NSFs, OSFs, or ISFs to the query BFs. Finally, we link the query BF to the target SF (which shares a more than 90% similarity with the query SF) as partial reuse. Compared with code search, OSS reuse detection also regards these ISFs as matching candidates.

**Vulnerability detection.** Similar to the situation of OSS reuse detection, a vulnerable SF, which is inlined to the BFs also spreads vulnerabilities to BFs. Thus, in vulnerability detection, we regard BFs with a vulnerable NSF, OSF, or ISF as vulnerable BFs that are needed to be detected.

**Patch presence test.** The labels in the patch presence test are given similar to vulnerability detection. We mark all the BFs with a patched NSF, OSF, or ISF as needed to be detected.

4 INVESTIGATION ON THE EXTENT OF FUNCTION INLINING

Before analyzing the extent of inlining, we first discover that inlining is not the same in all circumstances. Firstly, inlining is defined by the compiler. Thus, inlining is influenced by the compilation
settings. Secondly, the compiler decides whether to inline by considering the features of a call site, which will differ in different projects. Thus inlining also depends on the projects. Finally, we noticed that different applications also care about different parts of inlining. Thus, in this section, we propose three RQs to help understand the extent of function inlining.

4.1 RQ1.1: What Is the Extent of Function Inlining Under Different Compilations?

Figure 4 shows the statistics of function inlining in dataset I compiled by 8 architectures, 9 compilers, and 4 optimizations. Due to page limit, we only present results in clang-7.0 and gcc-8.2.0. In Figure 4, “Inlining ratio” shows the percent of BFIs in BFs, “ISFs/BFI” represents the number of ISFs that a BFI inlines, “BFIs/ISF” represents the number of BFIs that an ISF is inlined into. Here, we calculate these values by average. These metrics measure three aspects associated with function inlining: happening frequency in binary functions (Inlining Ratio), inlining degree in binary functions (ISFs/BFI), and inlining degree in source functions (BFIs/ISF).

4.1.1 Statistics in Default Ox Optimizations. For statistics of all 64 compilation options, we can see that the inline ratios range from 0.94% compiling with clang-7.0 in O1 to ARM64 to 41.67% compiling with gcc-8.2.0 in O3 to MIPS64. And when a binary function is generated with function inlining, at least 1.39 ISFs are inlined averagely. Moreover, according to our observation, a BFI may inline 200 different source functions to form the final binary function. When we turn the view to BFIs/ISF, an ISF will be inlined by at least 1.22 BFIs on average, and in some cases, an ISF may be inlined into 244 BFIs.

Statistics of different optimizations. For compilation settings from the three levels, we start with the most fundamental setting—optimizations. No matter what arch or compiler is, there is a straight increase in inlining ratios from low optimizations (i.e., O0, O1) to high optimizations (i.e., O2, O3). Same with the inlining ratios, a BFI tends to inline more ISFs, and an ISF tends to be inlined by more BFIs with higher optimizations. When applying the O3 option, more than 36% of

---

**Fig. 4. Function inlining statistics in different compilation settings.**
BFs will inline ISFs. As released binaries are often compiled under high optimizations to ensure their efficiency, we believe that function inlining is a common phenomenon.

**Statistics of different compilers.** The influence of different compilers is not directly pushed on generated binaries but through its inlining strategy to effect optimizations. An evident difference between GCC and Clang is that when compiling in O1, GCC produced a binary function with higher inlining ratios than Clang. This is because GCC applies “-finline-functions-called-once” when using O1. And the inlining ratio of GCC grows gradually from O0 to O3, while the ratios of Clang are divided into two parts: low optimizations and high optimizations. This is also caused by the inlining strategies of compilers. GCC gradually adds options to generate O0 to O3 while Clang uses “always inliner” pass in O0 and O1, and “inliner” pass. Besides, we find that compilers in the same family (in GCC or in Clang) with different versions have little inlining difference. Only that inlining conducted by GCC after 8.2.0 tends to have a little higher inlining ratio than before due to the modified run-time estimation metrics driving inliner heuristics.

**Statistics of different architectures.** When applying high optimizations, all 64-bit architectures have a little higher inlining frequency and degrees than their corresponding 32-bit architectures except X86. Among these 8 architectures, MIPS-64 has the highest inlining ratio, while ARM-64 has the highest inlining degree. Interestingly, these inlining differences in different architectures mostly come from the variability in C/C++ [40]. Preprocessor directives, such as `#define` and `#ifdef`, are typically used to make source programs easy to change and easy to compile in different execution environments [18]. When compiling a project under different architectures, different `#ifdef` directives are satisfied, leading different function content to take part in compilation, which then forms different function call graphs (FCGs) and finally makes the compiler conduct different inlining decisions.

**4.1.2 Statistics of using LTO.** Apart from the above compilation settings, we also noticed that LTO (Link-Time Optimizations) would also enable inlining across compilation units. LTO is a form of inter-procedural optimization that is performed at the time of linking application code. And default Ox optimizations are inter-procedural optimizations before linking time. LTO can be applied upon default Ox optimizations.

Figure 5 shows the inlining statistics in Coreutils v8.29 using Clang with and without LTO. We use abbreviations to represent the same metrics in Figure 4, such as IR for Inlining Ratio, B2S for ISFs/BFI, and S2B for BFIs/ISF. Compared with binaries without LTO, binaries enabled with LTO have 30% more inlining ratios. And when compiling in O3 enabled with LTO, more than 70% of the BFs are BFIs. Apart from the increasing frequency of inlining, two more ISFs are inlined into the BFIs when using LTO averages. The increasing frequency and extent of inlining bring more challenges to existing binary code similarity works.
Table 3. Function Inlining Statistics in Different Kinds of Projects

|       | D  | I   | II  | III | IV  |
|-------|----|-----|-----|-----|-----|
|       | Opt| O0  | O1  | O2  | O3  |
| IR    | 2.7%| 18.4%| 26.4%| 39.3%| 6.5%|
| B2S_avg| 2.35| 2.25| 2.27| 2.67| 1.86|
| B2S_max| 52  | 76  | 89  | 111 | 63  |
| S2B_avg| 1.41| 1.35| 1.33| 1.86| 2.99|
| S2B_max| 25  | 74  | 82  | 76  | 46  |

Answering RQ1.1: Inlining happens in high optimizations, and its ratio usually ranges from 30% to 40% when applying O3. However, inlining under different compilers may differ in the same optimizations. Besides, when combining O3 with LTO, inlining ratios can raise to 70%, and averagely a BFI can inline four ISFs.

4.2 RQ1.2: What Is the Extent of Function Inlining in Different Projects?

Different projects may have different inlining statistics due to their coding style and design rule. To avoid getting a biased understanding, we first analyze inlining statistics in four kinds of projects, including system code from dataset I, zip/unzip, timing, and data transformation code from dataset II, cryptography, database, and image processing code from dataset III, and browsers from dataset IV.

Table 3 shows the inlining statistics of these four datasets. Among the first three datasets, dataset II has the highest inlining frequency, and dataset III has the largest inlining degree. In particular, we notice that sometimes a BFI in dataset III can inline more than 2,000 ISFs, and an ISF can be inlined into more than 800 BFIs. This is caused by some extremely-long call chains in the cryptography module of OpenSSL [17], and we noticed that long call chains are also common in datasets I and II. And in dataset IV, we observe higher inlining ratios in the released binary compared with datasets I-III, which are compiled with default Ox optimizations, indicating large applications may use additional custom optimizations to further ensure the efficiency of the released executable.

Interestingly, chromium uses the following measure to further facilitate inlining [15]. Firstly, chromium uses profiling-based inlining heuristics to guide inlining. In detail, to decide which functions should be inlined, they first instrument the project code for profiling and run the instrumented executable in various scenarios. Then they recorded the resulting performance under different inlining decisions. Finally, using the resulting performance profile, they guide the inlining decisions of the compiler when compiling the released binaries. Secondly, chromium is applying LTO to further facilitate more inlining. Without LTO, compilers can only inline function calls in one single procedural. However, with LTO, inter-procedural function calls are also regarded as inlining candidates by the compiler. As a result, chromium conducts inlining on a large proportion of function calls and obtains a 57.9% inlining frequency in its release binary.

Answering RQ1.2: Generally speaking, inlining widely exists in different kinds of projects. In some projects with extremely-long call chains, such as OpenSSL, a binary function can inline thousands of functions. And in some large application binaries, due to some strategies such as profiling-guider inlining and LTO, inlining may happen more frequently.

4.3 RQ1.3: What Parts of Inlining Are Cared for in Different Applications?

Different applications care about different parts of inlining due to their objectives. For example, code search aims to find target functions with similar functionality to the query function; thus, it needs to search BFIs originated from the same OSF when function inlining happens. OSS reuse detection and vulnerability/patch detection aim to search functions containing the code of the reused or vulnerable functions; thus, they need to search BFIs containing the same ISFs. Here,
we focus on the inlining under OSS reuse detection and vulnerability/patch detection and analyze the existence and proportion of target SFs in the query BF.

As shown in Figure 6(a), we divided the relation between query BF and target SF into three classes: NSF, OSF, and ISF. For example, OSF means that the target SF is the OSF of the query BF, while ISF means that the target SF is an ISF of the query BF. Similar to Figure 6(a), in Figure 6(b), OSF means that query BF and target BF share the same OSF, while ISF means the NSF of target BF (which is an NBF) is an ISF of the query BF.

For OSS reuse detection, we noticed that when optimization goes from O0 to O3, ISF is gradually counting for the largest part, and NSF decreases from more than 60% to nearly 2%, which indicates the majority reuse between source projects will be inlined when applying high optimization, making it a new challenge for existing works.

For vulnerability/patch detection, when applying high optimization, OSF becomes the majority where NBFs and BFIs need to be matched. Besides, 30% of vulnerable functions are also inlined into BFIs, and finding these BFIs is also a task of existing works.

Answering RQ1.3: Code search works cares about the BFIs with the same OSF, while OSS reuse detection and vulnerability/patch detection also care about the BFIs with reused or vulnerable ISFs. The proportion of reused or vulnerable ISFs is not rare. Thus, they are also needed to be considered when conducting matching.

4.4 Answer to RQ1

In summary, function inlining widely exists in high optimizations and different projects. For released binaries, the inlining ratios range from 35% to more than 70%, and a BFI inlines 2–4 ISFs on average. Code search works should consider the unequal semantics between functions that inlining has brought. Besides, 80% of reused SFs and 30% of vulnerable SFs are inlined to BFIs when applying high optimization. OSS reuse detection and vulnerability/patch detection should also care about the hidden risky functions inlined into binary functions.

5 PERFORMANCE OF EXISTING WORKS UNDER INLINING

This section will carry out several evaluations on the performance of existing works under inlining. Directly, we will separately evaluate existing works in code search, OSS reuse detection, vulnerability detection, and patch presence test. To facilitate evaluation, we set different objectives for different applications. As we illustrated in Section 4.3, code search only needs to match BFs with the same functionalities, while OSS reuse detection and vulnerability/patch detection need to match every reused function. Thus, in our evaluation, apart from BFIs with the same OSF that code search works are required to accomplish, existing OSS reuse detection works and vulnerability/patch detection works are also evaluated in finding BFIs containing reused or vulnerable ISFs.

As inlining always comes with high optimizations, excluding the influence from other added optimization options is necessary. To evaluate the effect merely from inlining, we first select SFs that can produce BFs under all compilation settings. Then, we split these SFs into two classes: OSFs whose BFs compiled in high optimization are BFIs and NSFs whose BFs are all NBFs. The only difference for BFs generated from these two classes is whether inlining has happened. Thus evaluations conducted on these two split datasets only reflect the influence of inlining.

For all works selected for evaluation, we use its default setting in its code or its article and present its direct output.

5.1 Evaluation of Code Search Works

For code search works, we select CodeCMR [66] for binary2source code search and SAFE [46] for binary2binary code search. And, we run all the evaluations on dataset I. Here are the results:
5.1.1 Binary2source Code Search. We select CodeCMR because it is the state-of-the-art function level binary2source code searching work. And its application binaryai\footnote{https://binaryai.tencent.com/} has been released for public usage. We analyze the impact of inlining on the performance of binaryai at a downstream task of code search: retrieving similar SFs for the query BF.

**Cross-optimization evaluation for CodeCMR.** Figure 7(a) shows our evaluation of CodeCMR in BFIs and NBFs. We use the same metrics (recall@1 and recall@10) used in CodeCMR to evaluate its performance. As shown in Figure 7(a), blue lines present the performance of matching OSFs with their BFIs, and red lines present the performance of matching NSFs with their NBFs. When matching functions without inlining, we can see a smooth curve for both recall@1 and recall@10, though they suffered a subtle loss when matching NSFs with NBFs from O0 to O3. But when matching functions with inlining, we can see a drastic loss from O0 to O3, about 30% for recall@1 and 25% for recall@10, which indicates that function inlining has brought a great challenge to CodeCMR.

**Inlining-degree evaluation for CodeCMR.** To further analyze the influence of inlining on CodeCMR, we rearrange the dataset into classes by their inlining influence degrees. In detail, we calculate the ratio between the total length of ISFs and the length of the OSF to define the influence degree that inlining has on matching a BFI with an OSF. And for matching pairs with different inlining degrees, we calculate their average recall@1 and recall@10 to form Figure 7(b).

In Figure 7(b), blue lines show the change of recall@1 and red lines for the recall@10. Inlining influence degree as “1” represents matching pairs that ISFs of BFI have a 1× to 2× length equal to the OSF of BFI. As shown in Figure 7(b), both recall@1 and recall@10 decrease dramatically when inlining influence degree increase. In detail, recall@1 reaches nearly 80% when the ISFs have a smaller length than the OSF but falls directly to 0 when the total length of ISFs is larger than the length of OSF. Recall@10 decreases when the inlining influence degree ranges from 0 to 10 and experiences a fluctuation between 10 and 30. But in general, when the inlining influence degree is larger than 1, it has become particularly difficult for CodeCMR to find the correct OSFs.

5.1.2 Binary2binary Code Search. We select SAFE for code search as it can represent the common workflow of most recent learning-based methods: training a model to produce embedding, input functions to obtain embeddings, and comparing embeddings for similarities.

Figure 8 shows the function similarity that SAFE obtained for cross-optimization code search. To reveal the impact of function inlining, we divide the matching cases into normal cases, which only compares two NBF and inlining cases where at least one function is a BFI. In Figure 8, we use blue lines to represent the similarity in normal matching and red lines for inlining matching. Besides,
we use “G” for GCC and “C” for Clang. And x-coordinate shows the detailed optimization setting of the experiments. For example, the first O1 with “–O0–” below represents searching query BFs generated by O0 in target BFs generated by O1.

In general, SAFE suffers a 20% loss in similarity when detecting NBFs at the normal cross-optimization task (without inlining), but the loss increase to more than 40% when detecting BFIs (with inlining). Interestingly, we noticed that inlining in clang has a greater impact on SAFE than gcc, and this difference becomes more obvious when crossing O0-O1 to O2-O3.

5.2 Evaluation of OSS Reuse Detection Works

As CodeCMR is the state-of-the-art binary2source function matching works, we extend CodeCMR to detect partial reuses. To evaluate the effectiveness of CodeCMR on OSS reuse detection, we run CodeCMR on dataset II to see its results.

Table 4 shows the results of CodeCMR in detecting three types of reuse corresponding to three classes in Section 4.3. Figure 10 shows the distribution of the similarities of reused and irrelevant function pairs. When matching OSFs with BFIs, CodeCMR returns a similarity a little lower than matching NSF with NBFs. However, when searching ISFs for BFIs, CodeCMR can only return a similarity of less than 10% on average. As a result, CodeCMR can detect most NSF and OSF reused in NBFs and BFIs, but missed most ISFs that are inlined into BFIs.
5.3 Evaluation of Vulnerability Detection Works

We select Vulseeker for vulnerability detection. To see the effect of inlining, we recompile dataset III with “-fno-inline” to create a dataset without inlining (dataset III-NI). Then we use vulnerable functions in dataset III-NI to generate fingerprints and functions in dataset III as the target. Results are shown in three types corresponding to three classes in Section 4.3.

As shown in Table 5 and Figure 11, Vulseeker obtained a similarity of more than 90% for normal matching cases, a similarity of 80% when searching BFI, but a similarity of less than 65% when searching inlined functions. As there are many dissimilar function pairs sharing an approximate similarity with ISF pairs, BFIs, which inline vulnerable ISFs, are very hard to find.

5.4 Evaluation of Patch Presence Test Works

We select BinXray and dataset III for the patch presence test. As BinXray targets detecting patches at the same optimization, we conduct two experiments: default and cross-inlining. The default experiment generates fingerprints from binary functions compiled in O2 with inlining and search in binaries compiled in the same setting. The cross-inlining experiment generates fingerprints from binary functions compiled in O2 without inlining but searches binary functions compiled in O2 with inlining. Here are the results.

Figure 9 shows the results of default experiment and cross-inlining experiment. The number of rightly identified functions drops when detecting NBFs in the cross-inlining experiment compared with the default experiment, and this gap grows much larger for BFIs. Only about 2/5 of the vulnerable BFIs are rightly classified in the cross-inlining experiment, while 3/5 fall to “cannot analyze”.

Interestingly, although the rightly identified cases decrease dramatically, wrongly identified only increase a little. Instead, more cases become hard to be analyzed. That is because the analysis of BinXray relies on its successful identification of the patch boundary. For BFIs with vulnerable OSF that we mention in Section 4.3, BinXray cannot find a similar patch boundary derived from dataset III-NI.

Another problem is that BinXray regards the functions with the same name as the vulnerable function as the suspicious function to conduct detection. Thus, BFIs which has inlined vulnerable ISFs will all be missed. Note that these BFIs with vulnerable ISFs account for 30% when compiling in O3 as shown in Section 4.3.

5.5 Answering to RQ2

In summary, code search works suffer a 20%–30% performance loss when detecting functions with inlining. Ignorance of semantics from inlined functions reveals their inherent defect in detecting function similarities. Besides, OSS reuse detection and vulnerability/patch detection works cannot detect inlined functions. Thus, attackers or plagiarists can inline these risky functions into binary functions to bypass the detection.

6 EVALUATION OF EXISTING INLINING-SIMULATION STRATEGIES

In Section 5, we have shown that existing works suffer from the challenges that inlining brings. To resolve these challenges, existing works propose inlining-simulation strategies to simulate the inlining results of compilers. In this section, we will introduce existing inlining-simulation strategies and evaluate their effectiveness against function inlining in two fundamental tasks.

6.1 Introduction of Existing Inlining-Simulation Strategies

We first introduce the strategies used in two existing works: Bingo [22] and ASM2Vec [26].

ACM Transactions on Software Engineering and Methodology, Vol. 32, No. 4, Article 87. Pub. date: May 2023.
**Bingo.** Bingo proposes a selective inlining-simulation strategy to recursively expand callee functions for more precise similarity detection. To decide whether a callee should be inlined, Bingo listed six patterns for inlining conduction, and we summarized them into the following five cases where 2–5 are toward UD functions.

1. If the callee function is a library function, inline it.
2. If the UD callee function and its caller recursively call each other, inline it.
3. If the UD callee function only calls library functions, and more than half of library functions are not termination functions (e.g., exit and abort), inline it.
4. If the UD callee function only calls library functions, and more than half of library functions are termination functions, do not inline it.
5. If the UD callee function calls other UD functions, the inlining is decided by the following Equation (1).

   $$\alpha(f_c) = \frac{\text{outdegree}(f_c)}{\text{outdegree}(f_c) + \text{indegree}(f_c)}.$$  

   In Equation (1), in-degree and out-degree refer to the call relation between the callee \( f_c \) and other UD functions in the FCG. Bingo considers that the lower the value of \( \alpha(f_c) \), the more likely the callee should be inlined. Thus when \( \alpha(f_c) \) is lower than a threshold, this callee will be inlined. In Bingo’s original setting, this threshold is set to 0.01.

**ASM2Vec.** The inlining-simulation strategy of ASM2Vec employs the strategy of Bingo, but with some modifications and additional filters. On the one hand, ASM2Vec change the recursively inlining-simulation strategy into a one-layer inlining-simulation strategy, where they only expand the first-order callees. On the other hand, they defined a metric to remove lengthy callees, as shown in Equation (2).

   $$\delta(f_s, f_c) = \frac{\text{length}(f_c)}{\text{length}(f_s)}.$$  

   In Equation (2), \( f_c \) represents for the callee function and \( f_s \) for the caller. The length of the function is measured by the number of lines of instructions. When \( \delta \) is less than 0.6 or \( f_s \) consists of less than 10 lines of instructions (considered as a wrapper function), inlining will happen.

### 6.2 Evaluating Strategies on Finding Similar BFIs

As finding BFIs with the same OSF is the main objective of existing binary code similarity analysis works, we first evaluate their effectiveness in finding similar BFIs. According to their settings, the inlining-simulation strategies will be conducted for both the query BF and target BF, and the BFIs produced by the inlining-simulation strategies help find similar BFIs.

To evaluate their effectiveness, we implement the inlining-simulation strategies of Bingo and ASM2Vec and apply them to the dataset I. For each BF, Bingo and ASM2Vec will generate the BFIs using their inlining-simulation strategies. In detail, we use IDA Pro to disassemble binaries and extract features used for inlining selection. And, we provide two metrics to evaluate the effectiveness of their strategies.

\[
\text{cost} = BF_1' - BF_1 + BF_2' - BF_2,
\]

\[
\text{similarity} = \frac{BF_1' \cap BF_2'}{BF_1' \cup BF_2'}.
\]

In Equations (3) and (4), \( BF_1' \) and \( BF_2' \) are two BFIs that are need to be compared, and \( BF_1' \) and \( BF_2' \) are the BFIs’s generated by the inlining-simulation strategies proposed by Bingo and ASM2Vec. Equation (3) calculates the number of inlined BFIs for both \( BF_1 \) and \( BF_2 \), and Equation (4) calculates the similarity of produced \( BF_1' \) and \( BF_2' \). And we measure the BFIs in Equation (3) by the BFIs it contains and BFIs’ in Equation (4) by the SFs it contains.
Figure 12 shows the evaluation results of Bingo and ASM2Vec. To facilitate comparison, we also present the cost and similarity without applying inlining strategies (WI in Figure 12). The dotted line represents the similarity and the solid line for the cost. For example, the solid red line with the label “cost-Bingo” represents the cost when applying Bingo’s strategies. In general, compared with not applying strategies, Bingo averagely selects 4–5 BFs for inlining to obtain a 5% increase in its obtained similarity. ASM2Vec can reduce 0.25 BFs for inlining compared to Bingo while losing 1–2% in the similarity.

From the results, we can notice that Bingo and ASM2Vec are selective towards callees as recursively inlining all callees will cause a cost to inline nearly 55 callees while they only need to inline 4–5 callees. Their effectiveness is quite visible—a 5% increase in similarity. However, there is still a large space to improve, as when detecting functions cross O0 to O3, existing inlining-simulation strategies can only help guarantee that 80% of SFs in $BFI_1'$ and $BFI_2'$ are matched. And, we noticed that some BFs, which both are the callees of $BF_1$ and $BF_2$, are also inlined by $BFI_1'$ and $BFI_2'$, helping increase the function similarity.

In general, Bingo and ASM2Vec can help existing works improve their accuracy in finding similar BFs, but there is still a large space for improvement.

### 6.3 Evaluating Strategies on Finding Inlined Functions

In our evaluation of existing binary similarity analysis works, we noticed new requirements in some specific areas, including OSS reuse detection, vulnerability detection, and patch presence test, where inlined reused and inlined vulnerable functions also need to be detected. In this section, we will evaluate the ability of Bingo and ASM2Vec in finding inlined functions.

**Bingo.** To facilitate evaluation, we first apply Bingo to the dataset I and analyze its inlining decisions towards BFs. As Bingo has 3 cases (2, 3, 5) that callees (UD functions) will be inlined, to evaluate the effectiveness of its inlining-simulation strategies, we summarize the inlining decisions under the above three cases in O0-O3, which is shown in Table 6.
Investigating the Effect of Function Inlining on Binary Similarity Analysis

Fig. 13. Distribution of metrics for inlined source functions.

In Table 6, R, L, and α, respectively, represents inlining for Recursive call in case 2, inlining for callees with only Library call in case 3 and inlining determined by α in case 5. BFNN represents BFs that do not need to be inlined, while BFN represents BFs needed. To facilitate analysis, we only present results of comparing BFs generated by low optimization to BFs generated by high optimizations, and we analyze its inlining conduction for the BFs in low optimizations. For example, the first value 20.56% indicates that, when matching BFs generated by O0 to BFs generated by O1, among all inlined functions, 20.56% are BFs which are not needed to be inlined but inlined, as it falls to case 5 and its α satisfies Bingo’s requirements.

For an effective inlining-simulation strategy, the percentage of BFNN should be low as they do not need to be inlined, and the percentage of BFN should be high as they need to be inlined. But from Table 6, we can see that nearly 3/4 of inlined functions are BFNNs. To be more specific, among the three metrics, we noticed that only L had selected a large amount of BFNs while α and R rarely select BFNs. Bingo’s strategies waste a lot of cost in inlining unnecessary callees.

Besides, Table 6 also shows the coverage of needed inlined functions in each matching. When the compilation setting is not so diverse, Bingo can cover more than 70% inlined functions. But when comparing O0 to O3, 40% of inlined functions remain uncovered, making the results less complete.

ASM2Vec. ASM2Vec proposed two measures to reduce the functions Bingo chooses to inline. As shown in Table 6, they help reduce some unnecessary inlining candidates (see Section 6.2). However, their restrictions are still so strict that they would suffer additional misses of ISFs and obtain a lower coverage of inlined functions. To provide a comprehensive understanding of boundaries for inlining, we conduct a study to analyze inlining from SFs that comprise the final BFIs.

Figure 13(a) shows the distribution of depth for BFIs. We take the example in Figure 1(d) to illustrate the meaning of depth. In Figure 1(d), BFI netlink_sendmsg has inlined four source functions, including scm_set_cred and scm_send. And scm_send is called by OSF netlink_sendmsg and scm_set_cred is called by ISF scm_send, where we recognize the depth of scm_send as 1 and the depth of scm_set_cred as 2. Finally, we use the maximum of depths (i.e., 2) as the depth of this BFI. As shown in Figure 13(a), hundreds of thousands of BFIs inline ISFs with a calling depth of 2–3, and hundreds of BFIs inline ISFs even through 8 layers of the call graph. 30% of the BFIs have a depth of more than 1 layer.

Figure 13(b) shows the distributions of ISFs lengths. Though lots of ISFs have fewer than 10 lines, we noticed that tens of thousands of ISFs are above 100 lines. Even thousands of ISFs with a length above 1,000 are still inlined. Interestingly, the largest ISF is “aarch64_opcode_lookup_1” with 19083 lines of code which are inlined by the OSF “aarch64_opcode_lookup” with less than 10 lines of code. And this inlining happens during the compilation of Binutils compiling with GCC in O1. This is actually caused by the “-finline-functions-called-once” option as the ISF “aarch64_opcode_lookup_1” is only called by the OSF “aarch64_opcode_lookup”.

ACM Transactions on Software Engineering and Methodology, Vol. 32, No. 4, Article 87. Pub. date: May 2023.
Figure 13(c) shows the distribution of ratios between ISF and OSF. For all ISFs, we noticed that ratios of ISF/OSF range from 0 to more than 1,000. Only less than 1/3 of ISFs have a less than 0.6 of $\delta$ that will be inlined according to the ASM2Vec strategies, while 50% of the ISFs have a larger length than their OSF.

When using one-layer inlining and removing candidates whose $\delta >= 0.6$, ASM2Vec additionally misses ISFs. The missing of these ISFs not only influences the performance in finding the OSF of BFI but also its ability to detect BFIs that have inlined reused or vulnerable ISFs.

6.4 Answer to RQ3

Bingo and ASM2Vec can help find similar BFIs, but there is still a large space for promotion. When applying to detecting inlined functions, they miss nearly 40% of the inlined functions. More effective inlining-simulation strategies are needed to resolve the effect of function inlining on binary code analysis works.

7 TOWARDS MORE EFFECTIVE STRATEGIES

This section will discuss how to design a more effective inlining-simulation strategy to help further reduce the inlining cost and find more ISFs. In detail, we first recover the matching patterns that inlining brings, and based on the findings, we propose several suitable measures to help develop a more effective strategy.

7.1 Investigation of Matching Patterns Under Inlining

The matching pattern inlining brings to binary2source matching is “1-to-n” compared with the existing “1-to-1” matching mechanism. However, this becomes much more complex for binary2binary matching. Here, we summarize the patterns of function-level matching under function inlining, as shown in Figure 14.

1-to-1 matching. “1-to-1” matching is usually the pattern that most existing binary2binary work applies. Figure 14(a) just shows the matching without function inlining. And another matching with function inlining is also “1-to-1” matching, where two BFIs have inlined the same ISFs and thus have equal semantics.

1-to-n matching. “1-to-n” matching is similar to the pattern in binary2source matching. Figure 14(c) shows a similar matching with binary2source matching. And another matching with function inlining is also “1-to-1” matching, where two BFIs have inlined the same ISFs and thus have equal semantics.

n-to-n matching. “n-to-n” matching reveals the most complex matching patterns when conducting binary2binary matching. Figure 14(e) and (f) shows two classes of “n-to-n” matching. In detail, (e) shows two BFIs which have inlined different ISFs and the common semantics of these two BFIs only come from their shared OSF. (f) describe the BFI pairs which share the OSF and other ISFs.
Table 7. Distribution of binary2binary matching patterns

| Pattern | a       | b       | c       | d       | e       | f       |
|---------|---------|---------|---------|---------|---------|---------|
| Percent | 63.80%  | 6.33%   | 25.70%  | 3.11%   | 0.82%   | 0.21%   |

Existing inlining-simulation strategies are conducted both at the query and target functions, so they can resolve these three matching patterns. However, we recovered that not all these patterns are common in all matching cases. Thus, we could focus only on one or two particular patterns to resolve the challenges that inlining brings.

Table 7 shows the proportion of binary2binary matching patterns by comparing BFs originated from the same OSF but compiled with different architectures, compilers, and optimizations in dataset I. Interestingly, pattern e and f account for only 1% even with distant compilation settings. Instead, “1-to-1” matching accounts for the most common part as 70%, and “1-to-n” matching accounts for the rest 29%. To be more specific, comparing BFs compiled in low optimizations with BFs compiled in high optimizations are mostly cases falling to the pattern (c) and (d), which indicates the inlining is usually cumulative when optimization increases.

7.2 Suggestions for Designing More Effective Strategies

Considering the evaluation results of Bingo and ASM2Vec and the proportion of matching patterns in binary2binary matching, we find that existing strategies have three main shortcomings:

First, existing strategies conduct inlining for all cases on two sides. However, 70% of matching cases do not need inlining, while 29% only need inlining at one side. Only 1% of matching cases need inlining at two sides. Second, inlining unnecessary functions will bring difficulties in identifying the true match. For example, Bingo inlines some functions both in two functions, where different BFs with the same callees after inlining will also become similar. Third, many ISFs are missed as the side-effect of reducing the inlining candidates. For example, the restriction of Bingo makes it miss nearly 40% of inlined functions, and ASM2Vec applies two restrictions for candidates resulting in more missed ISFs. As a result, it cannot help detect BFs with vulnerable ISFs inlined. To overcome the shortcomings listed above, we propose several suggestions to reduce the inlining cost and help find the inlined functions.

**Necessary preprocessing to reduce candidates.** In binary2source matching, user-forced inlining candidates should be first identified. For example, functions with `__attribute__((always_inline))` will be inlined. Functions with `__attribute__((noinline))` will not be inlined. Thus, these functions can be determined whether to inline before applying strategies. Furthermore, callees of not inlined functions can be filtered, and callees of inlined ones can be added to candidates.

**Inspection of compilation settings to decide whether to apply strategies.** Leveraging some works designed for toolchain provenance recovery [21, 48, 55–57], the compilation setting of binaries can be inferred. Then, we could decide whether to apply strategies according to their compilation settings. For example, in binary2source matching, strategies are not needed if the binary is generated without inlining (such as “-O0”). In binary2binary matching, strategies are not needed if binaries are generated in the same compilation setting.

**Relaxing strategies of Bingo and ASM2Vec to recover more ISFs.** Section 6 reveals the effectiveness of Bingo and ASM2Vec in reducing candidates but at the cost of missing inlined functions. We find that the strategy of Bingo and ASM2Vec is too strict. We can relax their restriction to obtain more inlined functions. For example, changing the restriction from \( \alpha < 0.01 \) to \( \alpha < 0.5 \) helps increase the similarity by 10% while only inlining five more BFs and increasing the depth of BFs from 1 to 2 can improve the coverage of ISFs from 70% to nearly 93%.
Incremental inlining-simulation strategy to help locate ISFs. Considering that the most matching pattern is “1-to-n” when inlining happens, we do not need to expend all suitable callees at both sides. Interestingly, we find that we can conduct an inlining trail to filter inlining candidates if the method can distinguish between more similar and less similar function pairs. For function pairs such as the example in Figure 14(c), we named the NBF as A and the BFI as B. Initially, we discover many inlining candidates for both A and B. For A, we take a trail to inline one of the candidates into A to form A with inlining. Then, we compare the similarity between A and B to the similarity between A with inlining and B. If the similarity increase with inlining, we regard it as a suitable candidate and continue. Otherwise, the candidate, along with its callees, can be filtered. By conducting the inlining trail incrementally, we can quickly reduce the unnecessary branches and focus on some branches with deeper depth.

8 DISCUSSION

8.1 Study Settings

Inspecting inlining from the generated binaries. Some readers may wonder why we inspect inlining from the generated binaries instead of the compiler designs. There are three reasons. First, understanding the inlining strategies from compilers’ source code seems feasible, but this requires great manual effort. Moreover, as the inlining strategies of different compilers in different versions will differ, we regard directly extracting inlining statistics from binary as a more general way. Second, inlining results rely on not only the inlining strategies but also the features of the source project. For example, the inlining decision of GCC depends on the total cost of all inlined call-sites; thus, without the exact project, the inlining decision will not be clear. Third, we also combine some inlining strategies with the inlining statistics in binary. We think using strategies to assist in understanding function inlining is a good complement for inspecting function inlining in real binaries.

Metrics for evaluating existing works. In Section 5, we follow the following principles to determine the metrics for evaluating existing works. If the work has used a metric to evaluate its effectiveness, we directly use it. Otherwise, we use its direct output, such as the similarities between functions, to present its results. We do not choose to implement a new metric because the function similarities are enough to present their effectiveness under inlining, as the metrics used to evaluate downstream tasks are all based on function similarity.

Effect of function inlining on other kinds of binary similarity analysis works. In this work, we only evaluate the effect of function inlining on function-level binary analysis works. But, upon our observation, inlining also influences file-level methods, basic-block-level methods, and instruction-level methods. File-level methods [30–32] depend on FCG, and function inlining changes it, making the comparison less accurate. Function inlining also brings the basic blocks combination and instruction reorder crossing function borders, which affects basic-block-level methods and instruction-level methods [25, 26]. Besides, dynamic methods, such as [58–60], are less influenced, as the same input will be processed equally no matter whether inlining happens. We will leave the deep analysis of other kinds of binary analysis works as future works.

8.2 Effectiveness of Function Inlining Identification Method

Effectiveness of source entity identification tools. As our function inlining identification method is based on existing tools, its accuracy depends on the accuracy of existing tools. In fact, we notice that there are difficulties for understand in parsing source entities, so we use a conservative approach to label the source code. In detail, we run understand in its “strict” mode, where it will accurately identify the entity or add it to the unresolved list. And we manually label the unresolved entities to complement them.
Effectiveness of disassembling tools. As illustrated in some works [19, 49, 50], IDA Pro and other disassembling tools suffer from inaccuracies when attempting to discover function boundary. These inaccuracies will further affect the accuracy of our inlining identification method. However, the tools used in our method can be substituted. Some deep-learning-based disassembly frameworks, such as XDA [50], can be used to improve the accuracy of function boundary recovery.

Effectiveness of debugging tools. Currently, some works [43] have tested the debug information generated by compilers and found some code location problems and data value problems. For example, when the compiler applies dead code elimination, some code locations cannot be reached, and some values cannot be calculated. These issues have an effect on the completeness of line-level mapping but actually have little influence on our function inlining identification method. Considering that when a function is inlined to another function, there must be useful codes saved in the generated binary function. Thus, the function-level mappings that our works obtained will be more stable.

9 CONCLUSION
For the first time, our work investigates the effect of function inlining on binary similarity analysis research. Four datasets are constructed, and an automatic identification method is proposed to facilitate the analysis of function inlining. Our analysis finds that 36–70% binary functions have functions inlined in high optimizations, while most binary code similarity still regards function mapping as “1-to-1”. This mismatch causes 30% loss in performance during code search and 40% loss during vulnerability detection. Moreover, inlined functions are nearly all ignored during OSS reuse detection and patch presence test. Furthermore, existing inlining-simulation strategies usually produce less complete inlining results, leaving the binary similarity analysis under function inlining still challenged.

We propose several suggestions to design a more effective inlining-simulation strategy, but we still regard it as unfinished work. We hope that more researchers can pay attention to function inlining and conduct more research to resolve function-inlining-related issues.

REFERENCES
[1] 2009. Cisco settles FSF GPL lawsuit. [Online; accessed April 7, 2021]. Retrieved from http://arstechnica.com/information-technology/2009/05/cisco-settles-fsf-gpl-lawsuit-appoints-compliance-officer.
[2] 2015. VMware sued for failure to comply with Linux license. [Online; accessed April 7, 2021]. Retrieved from https://www.zdnet.com/article/vmware-sued-for-failure-to-comply-with-linuxs-license/.
[3] 2020. Reviews 1—SciTools. [Online; accessed 3-September-2021]. Retrieved from https://news.sophos.com/en-us/2020/04/26/asnarok/.
[4] 2020. What’s up, Emotet? CERT Polska. [Online; accessed 3-September-2021]. Retrieved from https://cert.pl/en/posts/2020/02/whats-up-emotet/.
[5] 2021. “Asnarök” Trojan targets firewalls. [Online; accessed 3-September-2021]. Retrieved from https://www.scitools.com/us-air-force-case-study.
[6] 2021. Binutils—GNU Project - Free Software Foundation. [Online; accessed 4-August-2021]. Retrieved from https://www.gnu.org/software/binutils/.
[7] 2021. Clang: A C language family frontend for LLVM. [Online; accessed 20-April-2021]. Retrieved from https://clang.llvm.org/.
[8] 2021. Coreutils—GNU core utilities. [Online; accessed 23-April-2021]. Retrieved from https://www.gnu.org/software/coreutils/.
[9] 2021. findutils—GNU Project - Free Software Foundation (FSF). [Online; accessed 4-August-2021]. Retrieved from https://www.gnu.org/software/findutils/.
[10] 2021. Function Inlining. [Online; accessed 3-September-2021]. Retrieved from https://compileroptimizations.com/category/function_inlining.htm.
[11] 2021. GCC, the GNU Compiler Collection. [Online; accessed 16-April-2021]. Retrieved from https://gcc.gnu.org/.
[12] 2021. The Chromium Projects. [Online; accessed 28-December-2021]. Retrieved from https://www.chromium.org/.
[13] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In Proceedings of the 25th USENIX Security Symposium, USENIX Security 16, Austin. USENIX Association, 583–600.

[20] 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.

[21] Sébastien Bardin, Tristan Benoit, and Jean-Yves Marion. 2021. Compiler and optimization level recognition using graph neural networks. In Proceedings of the MLPA 2020-Machine Learning for Program Analysis.

[22] 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.

[23] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016. ACM, 266–280. DOI: https://doi.org/10.1145/2908080.2908126

[24] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. Acm Sigplan Notices 49, 6 (2014), 349–360.

[25] Steven H. H. Ding, Benjamin C. M. 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.

[26] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 472–489.

[27] 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’17). ACM, 2169–2185. DOI: https://doi.org/10.1145/3133956.3134048

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

[29] 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 ({USENIX} Security 14). 303–317.

[30] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Transactions on Information Forensics and Security 13, 8 (2018), 1890–1905.

[31] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. 2017. Dapasa: Detecting android piggy-backed apps through sensitive subgraph analysis. IEEE Transactions on Information Forensics and Security 12, 8 (2017), 1772–1785.

[32] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. 2019. Graph embedding based familial analysis of android malware using unsupervised learning. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 771–782.

[33] 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 Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019. IEEE, 564–568. DOI: https://doi.org/10.1109/SANER.2019.8667977

[34] 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 Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019. IEEE, 1038–1049. DOI: https://doi.org/10.1109/ASE.2019.00100

[35] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. 2008. BinHunt: Automatically finding semantic differences in binary programs. In Proceedings of the Information and Communications Security, 10th International Conference, ICICS 2008 (Lecture Notes in Computer Science, Vol. 5308). Springer, 238–255. DOI: https://doi.org/10.1007/978-3-540-88625-9_16
Investigating the Effect of Function Inlining on Binary Similarity Analysis

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

[37] Dale Gardner. 2019. Technology Insight for Software Composition Analysis. [Online; accessed 27-November-2021]. Retrieved from https://www.gartner.com/en/documents/3971011/technology-insight-for-software-composition-analysis.

[38] 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). ACM, 63–72. DOI: https://doi.org/10.1145/1985441.1985453

[39] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating source-binary code similarity with graph triplet-loss network. In Proceedings of the 16th ACM ASIA Conference on Computer and Communications Security (ASIACCS).

[40] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. 2011. Partial preprocessing C code for variability analysis. In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. 127–136.

[41] Dongjin Kim, Seong-je Cho, Sangchul Han, Minkyu Park, and Ilsun You. 2014. Open source software detection using function-level static software birthmark. Journal of Internet Services and Information Security 4, 4 (2014), 25–37. DOI: https://doi.org/10.22667/JISIS.2014.11.31.025

[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2017. Cross-architecture bug search in binary executables. it Information Technology 59, 2 (2017), 83. DOI: https://doi.org/10.1515/itit-2016-0040

[43] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. 2019. A massive analysis of ethersum smart contracts empirical study and code metrics. IEEE Access 7, 1 (2019), 78194–78213.

[44] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 1052–1065.

[45] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. aff: Cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 667–678.

[46] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. IEEE Transactions on Software Engineering 43, 12 (2017), 1157–1177. DOI: https://doi.org/10.1109/TSE.2017.2655046

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

[48] Dhaval Miyani, Zhen Huang, and David Lie. 2017. BinPro: A tool for binary source code provenance. CoRR abs/1711.00830 (2017). Retrieved from https://arxiv.org/abs/1711.00830.

[49] Yuhe Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. 2020. o-classesx: Compiler provenance recovery with attention mechanism from a short code fragment. In Proceedings of the 3rd Workshop on Binary Analysis Research.

[50] 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 Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 833–851.

[51] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. In Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS’21). The Internet Society. https://www.ndss-symposium.org/ndss-paper/xda-accurate-robust-disassembly-with-transfer-learning/.

[52] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2017. Cross-architecture bug search in binary executables. it Information Technology 59, 2 (2017), 83. DOI: https://doi.org/10.1515/itit-2016-0040

[53] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. 2019. A massive analysis of ethersum smart contracts empirical study and code metrics. IEEE Access 7, 1 (2019), 78194–78213.

[54] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2000. JPlag: Finding plagiarisms among a set of programs. Citeseer.

[55] Ashkan Rahimian, Philippe Charland, Stere Preda, and Mourad Debbabi. 2012. RESource: A framework for online matching of assembly with open source code. In Proceedings of the Foundations and Practice of Security—5th International Symposium, FPS 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7743). Springer, 211–226. DOI: https://doi.org/10.1007/978-3-642-37119-6_14

[56] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. 2015. Bincomp: A stratified approach to compiler provenance attribution. Digital Investigation 14, Supplement 1 (2015), S146–S155.

[57] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In Proceedings of the 2011 International Symposium on Software Testing and Analysis. 100–110.
[57] Zhenzhou Tian, Yaqian Huang, Borun Xie, Yanping Chen, Lingwei Chen, and Dinghao Wu. 2021. Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access* 9, 1 (2021), 49160–49175.

[58] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2016. Exploiting thread-related system calls for plagiarism detection of multithreaded programs. *Journal of Systems and Software* 9, 1 (2016), 136–148.

[59] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Eryue Zhuang, Ming Fan, and Zijiang Yang. 2017. Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Transactions on Software Engineering* 44, 5 (2017), 491–511.

[60] Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2015. Software plagiarism detection with birthmarks based on dynamic key instruction sequences. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1217–1235.

[61] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.

[62] 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.

[63] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884.

[64] 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.

[65] 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.

[66] Zeping Yu, Wenxin Zheng, Jiachi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-modal retrieval for function-level binary source code matching. In *Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual*.

[67] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *Proceedings of the 27th USENIX Security Symposium, USENIX Security* 2018. USENIX Association, 887–902.

[68] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society.

Received 4 April 2022; revised 6 July 2022; accepted 24 August 2022