Published in Elsevier Journal of Systems and Software (JSS), 231(112603):1-20, January 2026.

FIN: Boosting Binary Code Embedding by Normalizing Function Inlinings

Mohammadhossein Amouei[a], Benjamin C. M. Fung[a,*], Philippe Charland

[a]*School of Information Studies, McGill University, Montreal, QC, Canada*
[b]*Mission Critical Cyber Security Section, Defence R&D Canada, Quebec, QC, Canada*

Abstract

Binary code similarity detection (BCSD) is essential for identifying similar code sections across different programs, regardless of their source languages, compilation options, or underlying architectures. It plays a crucial role in areas such as code plagiarism detection, malware analysis, and vulnerability discovery. However, BCSD faces significant challenges due to compiler optimizations, such as function inlining, which alter the binary structure. Existing rule-based function control flow graph (CFG) expansion strategies have limited success, due to low precision and recall in identifying inlined call sites. In this study, we present a detailed investigation of function inlining and propose an AI-driven solution to expand CFGs, offering improvements for BCSD approaches. We designed a set of features for a machine learning algorithm to identify functions at O0 and O1 optimizations that may be inlined at the higher optimizations O2 and O3, without prior knowledge of the optimization level. By utilizing this information to expand function CFGs, we observed significant enhancements in the performance of state-of-the-art binary code representation learning techniques. Experimental results show that our proposed method increases the effectiveness of representation learning approaches by up to 21.54%. Additionally, our experiments show that our proposed method can improve true positive rate in identifying known vulnerabilities.

Keywords: binary code, similarity detection, function inlining, control flow graph, random forest

*Corresponding author.
Email address: ben.fung@mcgill.ca (Benjamin C. M. Fung)

1. Introduction

Binary code similarity detection (BCSD) involves comparing different binary code sequences to identify identical or similar code sections within different programs, often irrespective of their source languages, compilation options, or underlying architectures. It plays a pivotal role in various fields, including code plagiarism detection (Luo et al., 2017), malware analysis (Li et al., 2021; Sun et al., 2023), and vulnerability discovery (Zhao et al., 2019; Yu et al., 2021; Luo et al., 2023).

By analyzing the binary codes of programs, BCSD can uncover instances where code has been copied or slightly modified to obscure its origins, even when source code is unavailable or has been deliberately obfuscated. This technique is widely used in digital forensics to investigate code plagiarism, where it can trace code reuse or copying, supporting intellectual property protection (Luo et al., 2017). In malware analysis, BCSD is instrumental in identifying and classifying malicious software by detecting similarities between new malware samples and known threats. This enables cybersecurity professionals to quickly recognize and respond to emerging threats, understand malware behaviors, and develop effective countermeasures (Li et al., 2021; Sun et al., 2023). Additionally, BCSD assists in identifying reused or modified vulnerable code across binaries, aiding in the identification of security flaws without requiring access to source code. This proactive approach improves the ability to detect and remediate potential security issues before they can be exploited, thereby strengthening overall software security (Zhao et al., 2019; Yu et al., 2021; Luo et al., 2023).

Two pieces of binary code are similar if they exhibit comparable structural, behavioral, or semantic characteristics, indicating they perform similar functions or operations despite potential differences in their representation or compilation. In the literature, similarities between binaries are categorized into three primary types: similar, identical, and equivalent. Identical binaries are exactly the same at the byte level, implying that the sequences of instructions, with no modification, are the same when disassembled. If two pieces of binary code have dissimilar syntax but provide identical functionality, they are considered equivalent (Haq and Caballero, 2021).

Identical functions are often easily detected using hash-based or exact match techniques. However, in practice, it is common for two binary functions, even when compiled from the same source code, to be equivalent in function, but not identical in form (Chandramohan et al., 2016). This is

mainly due to various factors, including differences in compiler optimizations, compiler versions, or the target architecture for which the code is compiled. These factors can lead to changes in instruction sequences, memory allocation, and other optimizations that alter the binary's form without affecting its functionality, resulting in significantly different binary outputs for the same source code (Li et al., 2023).

Function inlining is a compiler optimization technique, where the compiler replaces a function call with the actual code of the callee function. This process eliminates the overhead associated with calling a function, such as the call and return instructions, potentially making the code faster and more efficient (Theodoridis et al., 2022). However, function inlining significantly impacts BCSD, because when a callee function is inlined, its code is merged with the caller's code, making functions that perform the same tasks look different at the binary level (Jia et al., 2023).

Function inlining significantly challenges BCSD by altering the direct mapping between binary functions across different optimization levels. This transformation often results in complex "1-to-n" or even "n-to-n" mapping scenarios. Specifically, function inlining optimization can cause a single function at one optimization level to correspond to multiple functions at another optimization level. More critically, multiple functions at one optimization level may collectively map to multiple functions at a different level, further complicating the process of establishing direct correlations between functions. This deviation from the traditional "1-to-1" mapping complicates the detection of similar binary functions, as evidenced by statistical findings where the proportion of function inlining ranges from 30% to 40% under certain optimization levels, and can sometimes reach nearly 70%. The high rates of inlining cause significant mismatches during code search and vulnerability detection, leading to a decrease in code search accuracy of up to 30%, and a decline in vulnerability detection efficacy of 40% (Jia et al., 2023).

There are two main strategies for handling function inlining in BCSD: CFG expansion and detect-and-remove. Studies like BinGo (Chandramohan et al., 2016), Asm2Vec (Ding et al., 2019), and OpTrans (Sha et al., 2025) use CFG expansion, i.e., they explicitly inline certain callee functions post-compilation, to ensure that equivalent functions compiled at different optimization levels remain similar. In contrast, methods such as BINO (Binosi et al., 2023) and ReIFunc (Lin et al., 2024) locate the boundaries of inlined functions within a caller's body so that those regions can be later excised. However, once a compiler inlines a callee, subsequent optimizations,

including constant folding, dead-code elimination, and common subexpression elimination, tend to fuse caller and callee instructions so tightly that no clean subgraph may not remain to delete; any removal attempt risks dropping or corrupting fused code. Moreover, a false positive in boundary detection could cause remove-based approaches to delete semantically essential instructions, whereas CFG expansion, at worst, merely duplicates a callee's graph without ever erasing the original code.

Although CFG expansion techniques are safe, existing approaches employ manually defined huristics to expand function CFGs to improve cross-optimization BCSD. These approaches employ selective inlining strategies guided by manually tuned thresholds, such as function size limits, stack size, and coupling scores, to determine whether a callee should be inlined within a given function. While these heuristics help mitigate code-size explosion and maintain scalability, they fail to fully capture the nuanced inlining behaviors exhibited by modern compilers across different optimization levels. Consequently, they often miss functions that should be inlined or mistakenly include functions that should not, resulting in low precision and recall when identifying inlined callees. As a result, they offer only minimal improvements in the performance of BCSD approaches.

In this study, we investigate function inlining in detail and propose a solution that can intelligently expand function CFGs, improving existing BCSD approaches. For this purpose, we first analyzed function inlining and its effects on BCSD performance. We then designed a set of features that could be used in a machine learning algorithm to identify a significant portion of functions at O0 and O1 optimizations that were inlined at the higher optimizations O2 and O3, without knowing the current optimization level. Using this information, we expanded the functions' CFGs and examined the impact of these CFG expansions on the performance of state-of-the-art binary code representation learning techniques.

This work specifically focuses on the impact of compiler optimizations and function inlining decisions on BCSD, as these transformations often disrupt binary similarity detection pipelines. We evaluated our approach on x86 binaries generated from C/C++ programs, as they are widely used in reverse engineering and vulnerability detection research. While the current scope is limited to x86, the primary goal is to address function inlining challenges caused by compilers and optimization levels, and not architecture-specific variations. Evaluations on other architectures and languages are left for future work.

Our experiments show that 30.63% of functions include at least one call site inlined with both O2 and O3 optimization levels. They also demonstrate that our proposed method can increase the performance of state-of-the-art representation learning approaches in terms of Mean Reciprocal Rank (MRR) value by up to 21.54%. Moreover, our real-world vulnerability detection analysis shows that FIN can improve BCSD in vulnerability detection. In summary, the contributions of this study are as follows:

- We investigated the impact of function inlining on BCSD techniques to gain a clearer understanding of the problems it introduces and their extent. In addition, we conducted a detailed study on the effect of CFG expansion on BCSD performance. Our findings show that the presence of inlinined functions poses a significant challenge for BCSD. Nevertheless, CFG expansion can potentially enhance BCSD effectiveness by mitigating the issues introduced by function inlining optimization.

- We developed and proposed a set of features for caller-callee pairs that can predict compiler decisions regarding inlining at high optimization levels. Our analysis of these features revealed that, contrary to the common belief that function size and the in and out degrees of a callee function are the most critical factors for deciding CFG expansion, the average distance of a callee function from its callers is actually the most significant feature.

- We proposed the *Function Inlining Normalizer* (*FIN*), an AI-driven CFG expansion approach designed to enhance the effectiveness of binary code representation learning. FIN intelligently identifies and inlines call sites within a function to increase the similarity of functions across different optimization levels. This adjustment equalizes the contextual information across all four optimization levels, thereby improving the performance of language models used for binary function representation learning.

- We developed and publicly released a tool[1] designed to generate ground truth data for investigating issues stemming from function inlining in cross-optimization BCSD. Although this paper focuses on functions inlined at O_2 and O_3 for ground truth generation, the tool itself is not

[1] `https://github.com/McGill-DMaS/FIN`

limited to these optimization levels; it can compare any two binaries regardless of their optimization settings.

The rest of the paper is organized as follows: Section 2 explains our problem and motivation. Section 3 presents FIN, our proposed solution. Section 4 describes the empirical studies, research questions, answers, and experimental results. Section 5 discusses limitations, future research directions, and threats to validity. Section 6 reviews related work. Finally, Section 7 concludes the paper with key findings.

2. Problem and Motivation

Function inlining modifies assembly functions by integrating called functions (callee) directly into the caller function (caller), significantly affecting the structure of the generated code. At the O0 optimization level, most optimizations are disabled, including function inlining. This level focuses on reducing compilation time and preserving the structure of the source code for debugging purposes.

Starting with O1, basic optimizations are enabled. Function inlining at this level is limited. The compiler begins to perform simple optimizations that do not significantly increase compilation time. While the compiler may inline a few small, frequently-called functions, these particular inlinings are not guaranteed to persist at higher optimization levels. At this stage, the compiler begins to perform simple optimizations that do not significantly increase compilation time.

At the O2 optimization level, the compiler activates a broader set of optimizations aimed at improving performance without excessive compilation time. Function inlining becomes more aggressive, with the compiler inlining functions that are small or frequently called. This leads to more efficient code execution by reducing function call overhead.

At the highest optimization level, O3, the compiler applies all optimizations from O2 and additional ones that may increase compilation time and code size for potential performance gains. Function inlining is even more aggressive at this level, with the compiler inlining larger functions and those with more complex control flows to maximize performance.

2.1. Problem Definition

Variability in function inlining across optimization levels poses significant challenges for BCSD, especially when comparing binaries compiled at differ-

ent optimization levels. To understand the impact of inlining across different optimization levels, consider the set of functions reachable from f through function calls at optimization level O. Let $G_O(f)$ denote this set, which includes all functions that are called, directly or indirectly, from f at that optimization level.

At optimization level O0, since no inlining occurs, $G_{O0}(f)$ represents the complete set of functions that f can call, matching the call graph derived from the source code. As optimization levels increase to O1, O2, and O3, the compiler may inline some of these functions into f. When a function g is inlined into f, the call to g within f is replaced by the whole or part of g's body. It is worth noting that g may still exist in $G_O(f)$ if there are still other direct or indirect calls to g that were not inlined.

At $O0$, f and the functions in $G_{O0}(f)$ remain separate entities. As the optimization level increases, some or all functions from $G_{O0}(f)$ may be inlined into f, altering f's structure. For a given optimization level $O \neq O0$, the transformed f can be conceptually represented as $\{f\} \cup G'$, where $G' \subseteq G_{O0}(f)$ consists of the functions inlined into f at O. This creates a 1-to-n mapping challenge: a single function f at O0 corresponds to f and potentially multiple functions G' that were merged into it at higher optimization levels. It is worth noting that we exclude function calls introduced by other compiler optimizations, such as replacing loops with calls to `memset`, if these functions do not exist in $G_{O0}(f)$. This is because a function that does not exist in the call graph at O0 cannot be inlined at $O0$.

Similarly, the n-to-n problem arises when multiple functions at one optimization level correspond to multiple functions at another level, with complex inlining relationships complicating direct correspondence. These discrepancies make it challenging to match functions across optimization levels. For example, at optimization level $O1$, the compiler may inline a subset of functions from $G_{O0}(f)$ into f, resulting in $\{f\} \cup G'_1$, where $G'_1 \subseteq G_{O0}(f)$. At another optimization level $O3$, a different subset $G'_3 \subseteq G_{O0}(f)$ may be inlined into f, yielding $\{f\} \cup G'_3$. The subsets G'_1 and G'_3 may or may not be equal. Consequently, when comparing f_{O1} and f_{O3}, we are effectively comparing $\{f\} \cup G'_1$ with $\{f\} \cup G'_3$. Since G'_1 and G'_3 may include different functions, the code structures of f_{O1} and f_{O3} differ, complicating direct comparisons and creating a many-to-many mapping challenge.

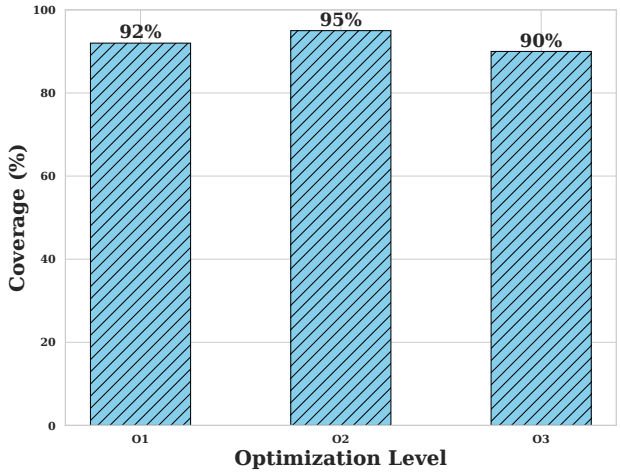


Figure 1: Percentage of caller–callee pairs that, when inlined at any optimization level (O1, O2, or O3), are also inlined at both O2 and O3 for GCC and Clang compilers.

2.2. Motivation and Core Idea

Our motivation stems from the observation that detecting similarities between binaries across the O2 and O3 optimization levels is generally easier, since these levels share very similar optimization flags and yield the highest recall rates for cross-optimization comparisons (Ding et al., 2019). Consequently, the sets of callee functions chosen for inlining at O2 and O3 tend to overlap significantly. In other words, the functions inlined at both levels follow a consistent pattern that an AI model can potentially learn. We further compared, for both GCC and Clang, the set of functions inlined at each optimization level to those inlined simultaneously at O2 and O3. Fig. 1 shows that over 90% of caller–callee pairs with an inlining relationship (i.e., where the callee was inlined into the caller at least once) at any optimization level (O1, O2, or O3) also exhibited that relationship at both O2 and O3. As expected, since O0 disables inlining, all relationships present at both O2 and O3 were absent at $O0$. At O1, 58% of these relationships were missing for GCC, while only 1.4% were missing for Clang. These observations suggest that normalizing binary functions by their inlining patterns and employing the set of functions inlined at both O2 and O3 may enhance BCSD, particularly in comparisons between O0 binaries and those produced at higher optimization levels.

To address the challenges posed by function inlining, we propose defin-

ing a synthetic optimization level $Onorm$ that normalizes inlining decisions across functions compiled at different optimization levels. Specifically, we aim to adjust functions compiled at lower optimization levels (O0 and O1) by inlining certain callees that are consistently inlined at higher optimization levels (O2 and O3). By focusing on the intersection of O2 and O3 we aim to increase similarity across optimization levels with minimal modifications rather than relying on blanket aggressive inlining. Over-inlining can rapidly inflate function sizes and control-flow complexity, complicating binary clone detection. In addition, modern language models generally have a fixed token limit; if a function becomes too large, the model may truncate or omit key instructions, thereby degrading the quality of the learned representations.

Let's define $I_O(f)$ as the set of functions that are inlined into f at optimization level O. Then, define:

$$I_{Onorm}(f) = I_{O2}(f) \cap I_{O3}(f)$$

This set $I_{Onorm}(f)$ represents the callees that are consistently inlined into f at both O2 and O3. By inlining these callees into f at every call site, we aim to achieve a more similar distribution of information within the function. To accomplish this, we define an adjusted function f_{Onorm} as follows:

$$f_{Onorm} = \text{Inline}(f, I_{Onorm}(f))$$

Here, $\text{Inline}(f, I_{Onorm}(f))$ denotes the process of inlining the callees in $I_{Onorm}(f)$ into f. Functions compiled at O2 or O3 will still be modified if some call sites to the callees in $I_{Onorm}(f)$ remain (i.e., only partial inlining has been applied).

By normalizing the inlining of certain functions across different optimization levels, we aim to reduce discrepancies in their underlying semantics, thereby improving the effectiveness of BCSD when comparing functions compiled at different optimization levels.

The rationale for focusing on caller–callee pairs rather than call site-level decisions stems from the added complexity that fine-grained inlining entails. Partial inlining patterns at higher optimization levels mean that some call sites may be inlined while others are not, which would require including call site-specific features. This approach can increase false negatives (e.g., predicting inlining at a less informative site, such as those located outside a language model's maximum context window, while missing a more critical one). Modern NLP models operate within fixed context windows

and may truncate inputs; if the first inlined site would fit, but a later one would not, inlining the latter yields no benefit. By treating each callee as a single unit and inlining it unrecursively at all available sites, we simplify the prediction task and ensure that the most semantically relevant code is consistently included for our representation-learning models.

However, determining $I_{Onorm}(f)$ is still challenging because, without debug information, we cannot directly observe the compiler's inlining decisions. These decisions are based on heuristics and vary depending on factors such as function size, complexity, and compiler settings. To overcome this, we hypothesize that it is possible to predict the compiler's inlining decisions using machine learning techniques. By extracting relevant features from the code and training models on known inlining decisions, our aim is to estimate $I_{Onorm}(f)$ even without knowing the specific optimization level used during compilation.

3. Approach

Fig. 2 presents the workflow of the inference stage (after training) for our proposed approach, FIN. During the training phase, we first generate ground-truth data by analyzing compiler inlining decisions across optimization levels (detailed in Section 3.1). This ground-truth guides the training of a random forest classifier to predict inlining decisions. In the inference stage, the method begins with disassembling binary code to extract assembly functions. Next, callees are identified for each function, and the features are computed for both the functions and their corresponding callees to generate the feature vectors (Section 3.2). These vectors are used by the trained classifier to predict whether function inlining should occur (Section 3.3). Finally, based on predictions, the CFGs are expanded to unify the optimization levels and improve BCSD (Section 3.4). The adjusted functions are subsequently passed into representation learning approaches for further processing.

3.1. Ground-truth Generation

In this study, we utilized the BinKit (Kim et al., 2022) dataset, a benchmark for binary code similarity analysis, which contains 51 GNU project (Project, 2024) packages. These packages were compiled using two compilers, GCC and Clang, across ten versions each, involving six optimization levels, and targeting eight different architectures. For our analysis, we specifically focused on packages compiled with gcc-11.2.0 and clang-13.0.0, restricting

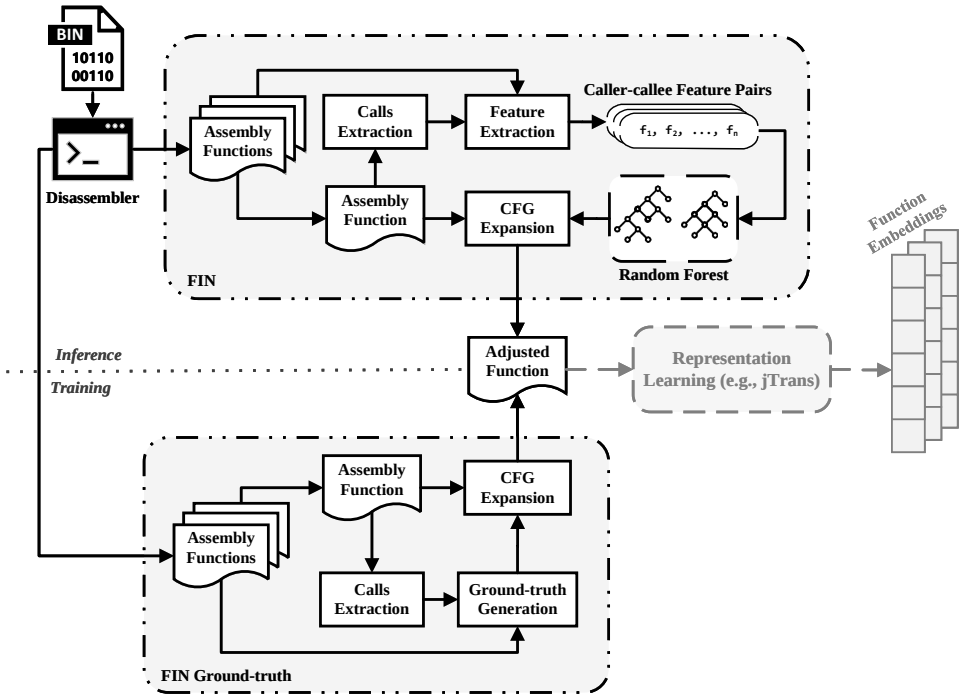


Figure 2: Overview of the FIN method.

our study to optimization levels O0, O1, O2, and O3 for the x86_64 architecture. More details are provided in Section 4.1.2.

Considering that the Binkit dataset comes pre-compiled with the *-g* option, it utilizes Dwarf debugging information to generate the *.debug_line* section in the binaries. This section is important, as it includes mappings from instructions to source code locations. Instruction-to-source mapping can be extracted using the pyelftools (Bendersky, 2025) tool.

To generate a ground-truth dataset, we first established equivalence between functions compiled at different optimization levels. For this purpose, we adopted the approach proposed in (Kim et al., 2022). According to this strategy, two functions are considered equivalent if they have the same name in their binaries, originated from the same source file, and share the same line numbers in the source code. Additionally, we verified that both functions belong to the same package to further reinforce their equivalence.

Next, we utilized instruction-to-source mappings to discover inlined func-

tions. Although instruction-to-source mappings can be influenced by compiler optimizations, research suggests that this information remains sufficiently reliable for accurately identifying inlined functions (Jia et al., 2023). Thus, we used this information to identify inlined functions at the O2 and O3 optimization levels. More specifically, we first extracted mappings for all callee functions associated with callers at the O0 optimization level. Then, these mappings were then compared against the mapping of the same caller functions at optimization levels O2 and O3. If any segment of the callee function's mappings matched those of the caller functions in O2 and O3, we inferred that the callee function was inlined at least once within the caller at these optimization levels. Consequently, such caller-callee pairs were marked with a label of 1, indicating inlining occurrence. However, we only consider a callee as inlined only if it undergoes inlining at both the O2 and O3 optimization levels. This approach is grounded in the understanding that comparing functions across these particular optimization levels is the easiest (Ding et al., 2019), leading to the hypothesis that the impact of differences in inlining between these levels is likely to be negligible.

The proposed method identifies callee functions rather than individual call sites for inlining, as this approach ensures that the expanded semantics of a function are incorporated into the broader dataset, even if the function is not fully inlined at all call sites of the caller. This allows the model to learn from a richer set of semantic features, enhancing its ability to generalize. However, this method can result in cases where functions marked as inlined at O2 and O3 are still partially present at certain call sites, requiring those functions to be further expanded during the normalization. This further expansion is needed because, except for the rare case of recursive calls, we are already inlining all corresponding call sites at O0 and O1. Therefore, extending this consistency to O2 and O3 is beneficial, as it aligns the representation of these functions across all considered optimization levels. By inlining these functions at every call site within the caller, we aim to achieve a more uniform and comparable distribution of information within the functions. However, we observed that this approach affected less than 2% of training caller functions (using ground-truth labels) and less than 4% of testing caller functions (using predicted labels), suggesting that its impact on the dataset's overall structure at O2 and O3 optimization levels is minimal.

Fig. 3 illustrates an example of function inlining detection at the O3 optimization level. In this case, the O0 version of the *file_existsp* function contains several function calls. We begin by extracting the instruction-source

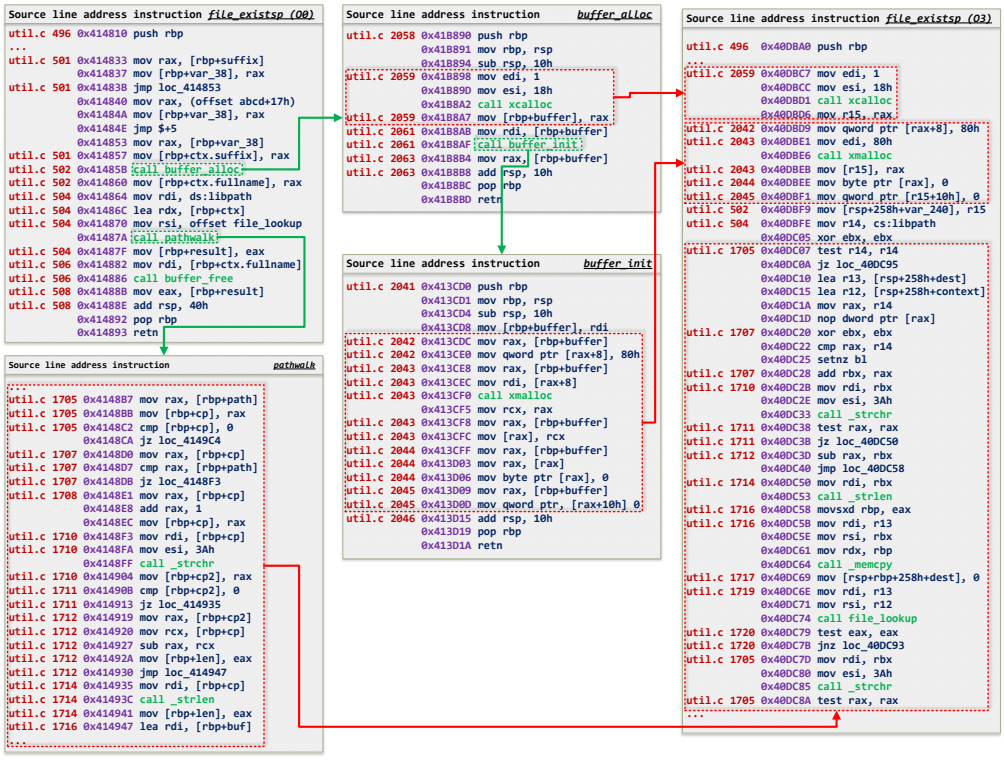


Figure 3: Illustration of Function Inlining Detection Process for Creating Ground Truth Data.

mapping for these callee functions, then search for these mappings in the O3 version of *file_existsp*. Notably, a portion of the *buffer_alloc* function corresponds to line 2059 in *util.c*, a mapping that persists in the O3 version of *file_existsp*. This observation leads us to conclude that the compiler has inlined *buffer_alloc* within *file_existsp* at the O3 level. We proceed to apply this method recursively for calls within callee functions until no further inlined functions are detected. For instance, *buffer_alloc* itself invokes *buffer_init*, which is also found to be inlined in the O3 version of *file_existsp*, demonstrating the depth of this inlining detection process.

3.2. Feature Extraction

Determining whether to inline a function presents a complex decision-making challenge. Compilers must carefully weigh the potential performance benefits of inlining against the associated trade-offs, such as increased code

size and other costs (Zhao and Amaral, 2004). While specific heuristics and internal metrics vary across compilers (e.g., GCC, Clang/LLVM, MSVC), several common factors are generally taken into account in this process:

- **Function Size:** Small functions, such as simple getters, setters, or basic arithmetic operations, are ideal candidates for inlining, as their runtime cost is dominated by call overhead. In contrast, larger functions are generally less suitable for inlining due to the potential for significant code size expansion with limited performance gains.

- **Function Complexity:** Functions with high complexity, characterized by numerous branches, loops, or nested function calls, are generally less likely to be inlined due to the significant increase in code size and the reduced likelihood of substantial performance benefits. Additionally, recursive functions are typically not inlined, except in specific cases such as tail recursion, which can be transformed into an iterative loop by the compiler to optimize performance.

- **Function Call Frequency and Profile-Guided Information:** Compilers often estimate the frequency of function calls through static analysis. For instance, calls within loops or those that occur multiple times are classified as "hot" and are deemed more advantageous for inlining. When profile-guided optimization (PGO) is employed, the compiler leverages runtime execution profiles to identify call sites that are frequently executed. These "hot" call sites are assigned a higher priority for inlining, as eliminating their call overhead can result in significant performance improvements.

In designing our feature set, we sought to encapsulate the multifaceted factors influencing compiler decisions by incorporating a diverse range of metrics. These include direct size measures, relative indicators, control-flow complexity assessments, and low-level operand usage profiles. Each feature was carefully selected to approximate known or inferred compiler heuristics. Below, we provide a detailed introduction to each feature, highlighting its rationale and how it contributes to modeling the compiler's decision-making process.

3.2.1. FuncSize (Absolute Function Size in Bytes)

This metric records the raw size of a function in machine code bytes. Inlining smaller functions typically delivers more pronounced benefits because

the call overhead often constitutes a significant portion of their runtime cost. Conversely, inlining large functions can incur significant code bloat with limited performance gains.

3.2.2. LoopCount (Number of Loops)

Loops significantly impact a function's structural complexity. Functions with numerous loops not only have a larger code footprint but also exhibit intricate control flows, complicating downstream optimization passes after inlining. Compilers typically employ thresholds and heuristics to avoid inlining functions with excessive loops unless there are compelling benefits against potential drawbacks like code size increase and cache inefficiency.

3.2.3. IsRecursive (Recursion Indicator)

Recursion is generally resistant to straightforward inlining, as it does not resolve the recursive pattern without additional transformations. Unless specific cases, such as tail recursion, are identified and optimized into iterative loops, repeatedly inlining a recursive function can result in unbounded code growth with minimal performance gains. Compilers typically adopt a conservative approach, avoiding the inlining of recursive functions unless further analysis supports a bounded unrolling strategy.

To capture this behavior, we include the `IsRecursive` feature in our model. This feature is a binary indicator, where a value of 1 denotes that the function is recursive, and a value of 0 indicates that the function is non-recursive. By incorporating IsRecursive, we align with the compiler's cautious stance on inlining recursive functions, reflecting its consideration of the associated risks and limitations.

3.2.4. SizeInc (Relative Increase in Program Size) and BBInc (Relative Increase in Basic Blocks)

Inlining eliminates the overhead associated with a function call but can duplicate the callee's instructions, potentially increasing the overall code size. To emulate the compiler's decision-making process, which balances the benefits of reduced call overhead against the global impact of code expansion, we introduce two features: SizeInc and BBInc.

- ***SizeInc*** quantifies the expected increase in code size by measuring the number of instructions that would be added to the caller if the callee were inlined.

- ***BBInc*** estimates the growth in the number of basic blocks resulting from inlining, capturing the structural expansion of the control flow graph.

To compute `SizeInc` and `BBInc`, we utilize the relative change formula:

$$\frac{x_2 - x_1}{x_1},$$

where x_2 denotes the program's size in bytes or the updated count of basic blocks following the inlining of a specific function, and x_1 refers to the program's size or the number of basic blocks before the inlining process. To calculate x_2, we use the following equation:

$$x_2 = x_1 - x' + (\lambda \times x') = x_1 + ((\lambda - 1) \times x'),$$

where, depending on the relative change we aim to calculate, x' represents either the callee function's size in bytes or its total number of basic blocks, while λ denotes the number of incoming calls to the function. We subtract one from λ under the assumption that the function is inlined at all call sites and as a result, would no longer be present within the program. Substituting x_2 into the relative change formula yields the following equation:

$$\Delta = \frac{(\lambda - 1) \times x'}{x_1}$$

We calculated Δ_s and Δ_b, denoting `SizeInc` and `BBInc`, respectively, and incorporated them into the feature set.

3.2.5. Z-Scores (SizeZScore, BBZScore, InCallsZScore, OutCallsZScore)

While absolute values (e.g., raw function size) provide a baseline, compilers often judge the characteristics of a function relative to the entire program (Theodoridis et al., 2022). Functions that deviate significantly from the norm may trigger special heuristics. To emulate this, we used standardized scores (z-scores):

$$z = \frac{X - \mu}{\sigma},$$

where z represents the z-score, X denotes the feature's value, μ signifies the mean of the feature across the program, and σ stands for the standard deviation of the feature within the program. We calculated z-score for four key features:

- *SizeZScore:* Quantifies how a function's size compares to the average function size within the program, expressed in terms of standard deviations from the mean. This metric reflects the extent to which a function is unusually large or small relative to its peers. Compilers often employ size-related thresholds when making inlining decisions; functions that are several standard deviations above the mean may exceed these cutoffs and be considered unsuitable for inlining.

- *BBZScore:* Measures how a function's basic block count deviates from the program-wide average, expressed as standard deviations from the mean. This metric highlights whether the function's control flow complexity is atypical. Functions with significantly higher basic block counts often involve intricate logic, which compilers typically approach cautiously when evaluating them for inline expansion.

- *InCallsZScore:* Captures the relative frequency with which a function is called, expressed in terms of standard deviations from the program-wide mean. This metric highlights functions that are invoked significantly more often than others. Functions with unusually high call frequencies are strong candidates for inlining at hot call sites, as eliminating the repetitive overhead of function calls can transform it into a one-time cost of additional instructions.

- *OutCallsZScore:* Quantifies the extent to which a function is call-heavy, measured as standard deviations from the program-wide average number of calls made by functions. This metric indicates whether a function invokes significantly more functions than is typical. Inlining call-heavy functions into a caller can lead to significant code expansion. This is because inlining replaces the function call with the function's body, and if that body contains multiple calls, each of those may also need to be inlined or managed, increasing the overall code size. Despite the potential benefit of eliminating a single call overhead, such functions are generally less appealing for inlining.

These z-scores contextualize each function's characteristics within the broader scope of the program, emulating the compiler's dynamic adaptation of heuristics based on global program statistics. By incorporating these relative measures, our approach aligns with how compilers leverage global insights to refine inlining decisions and prioritize functions for further optimization.

3.2.6. Operand Type Frequencies (o_reg, o_mem, o_phrase, o_displ, o_imm, o_near, o_fpreg)

Compilers take into account not only high-level metrics such as function size and loop counts, but also the "texture" of a function at the instruction level. The variability and frequency of different types of operands (Hex-Rays, 2024b) used can provide valuable insight into how data is accessed and manipulated within the function.

- Register Operands (*o_reg*): A high usage of register operations suggests a function that could benefit from inlining. Post-inlining, the compiler's register allocation optimizations can reduce instruction count further and improve execution throughput by minimizing memory accesses and leveraging faster register-based computations.

- Memory References (*o_mem*, *o_phrase*, *o_displ*): These features capture different forms of memory addressing within a function. Functions with a high prevalence of memory-bound instructions may pose challenges for optimization and register allocation when inlined. Such instructions can introduce additional complexity, especially if they do not simplify or integrate efficiently within the caller's context.

- Immediate Values (*o_imm*): Immediate values within a function often facilitate optimizations such as constant propagation and constant folding when the function is inlined. These transformations can lead to further simplifications and efficiency gains, making functions with a high occurrence of immediate values more favorable for inlining.

- Near Addresses (*o_near*): Near address references, such as relative jumps or calls, signify control-flow complexity within a function. Inlining functions with numerous near address references may simplify some control-flow structures by folding them into the caller. However, excessive control-flow complexity can impede further analysis and optimization, making such functions less attractive for inlining in certain contexts.

- Floating Point Register (*o_fpreg*): The use of floating-point registers and associated computations requires specialized handling and optimization. Inlining such functions may unlock opportunities for vectorization or improved floating-point instruction scheduling, depending on the caller's context.

The distribution and frequency of operand types within a function offer valuable indicators of its suitability for inlining. These features can reveal the function's complexity, optimization potential, and inlining overhead. Therefore, by integrating these features, we aim to approximate the way compilers incorporate low-level code patterns into their cost models.

3.2.7. Average Distance to Callers (AvgCallDist)

The average distance between a function and its callers significantly impacts how efficiently the CPU fetches and executes the function's instructions. When the caller and callee are far apart in memory, the processor is more likely to encounter instruction cache misses and pipeline stalls, harming performance (Chen and Chung, 2022). Functions frequently invoked from distant code regions may benefit from being moved closer to their callers or inlined to improve cache locality and reduce overhead. Compilers and linkers address this through code layout optimizations to identify frequently interacting functions and place them in adjacent memory regions to minimize caller-callee distance (Chen and Chung, 2022).

At lower optimization levels, such as O0 or O1, a function's natural proximity to its callers often reflects characteristics like small size, localized usage, and simple call relationships, making it a good candidate for inlining. While these optimization levels apply minimal transformations, such traits align with what compilers prioritize for inlining at higher levels like O2 or O3. Thus, we identified spatial relationships between functions as a valuable feature for predicting inlining decisions.

To build the feature vectors, we first extract the described features for both the caller and the callee in each caller-callee pair. This approach reflects a fundamental aspect of inlining decisions: both the caller's and the callee's characteristics influence the potential benefits and costs of integrating one function into another.

Callee attributes help estimate risks of code bloat, changes in complexity, and optimization opportunities introduced by inlining. However, the caller's profile is equally critical. For instance, a caller that is already large or complex may not be a suitable environment for inlining additional code, even if the callee appears ideal (Zhao and Amaral, 2004). Conversely, a structurally simple and non-dense caller can more easily absorb a callee's instructions without incurring significant penalties.

By extracting the same set of features, such as size, complexity indicators, and operand type frequencies, for both the caller and the callee, we approx-

imate the compiler's holistic evaluation of the call-callee pair. Through this approach, we aim to train a model that considers not only the callee's suitability for inlining but also whether the caller provides a conducive context for embedding the callee's logic.

Table 1: Average feature values for caller and callee, under GCC and Clang (Class 0 vs. Class 1)

Feature	**GCC**				**Clang**			
	Caller		**Callee**		**Caller**		**Callee**	
	Class 0	**Class 1**	**Class 0**	**Class 1**	**Class 0**	**Class 1**	**Class 0**	**Class 1**
FuncSize (KB)	2.0164	2.4514	0.4605	0.1749	1.8040	2.0316	0.5040	0.1613
AvgCallDist (KB)	56.2802	61.9754	171.0332	10.0488	49.6553	51.1279	187.8687	9.3681
LoopCount	0.0004	0.0005	0.0008	0.0015	0.0028	0.0021	0.0014	0.0006
SizeInc (%)	0.37	0.68	0.87	0.04	0.40	0.67	1.02	0.04
BBInc	0.39	0.70	0.87	0.04	0.0036	0.0059	0.0099	0.0004
SizeZScore	1.4389	1.7795	0.0396	-0.1289	1.2557	1.5669	0.0691	-0.1659
BBZscore	1.4381	1.7560	0.0463	-0.1436	1.3074	1.6124	0.0685	-0.1800
InCallsZScore	-0.0590	-0.0664	2.9220	0.0778	-0.0575	-0.0771	3.0756	0.1002
OutCallsZScore	1.3432	1.9695	0.0608	-0.1537	1.2603	1.3044	0.0808	-0.1851
IsRecursive	0.0607	0.0750	0.0349	0.0014	0.0653	0.0815	0.0469	0.0014
o_reg	361.0589	424.5251	87.2807	51.2832	408.3902	477.0281	97.5262	43.1628
o_mem	4.4020	5.2916	1.2723	0.8134	6.2861	7.7381	1.5145	0.6180
o_phrase	13.5424	16.3894	3.7727	1.9959	19.4364	27.6524	5.1402	1.8905
o_displ	147.4741	159.3307	28.2359	21.9068	127.0596	132.6433	34.2130	22.3549
o_imm	77.3868	81.1916	18.2039	9.4215	83.5391	104.2657	21.1404	9.7377
o_near	93.2617	106.2236	19.4754	8.5513	97.5166	109.6421	22.7912	8.3419
o_fpreg	0.0202	0.1433	0.0081	0.0618	0.0111	0.1005	0.0118	0.0240

Table 1 presents the mean value of each feature for both classes 0 and 1. To evaluate the effectiveness of our proposed features, we conducted a Wilcoxon signed-rank test on our train set at O0 and O1 (see Section 4.1.2) to compare feature distributions between class 0 (cases where the callee was not inlined at higher optimization levels) and class 1 (cases where the callee was inlined at higher optimization levels). Table 2 summarizes the mean percentage differences in feature values for samples in class 1 relative to those

Table 2: Means of feature values for samples in class 1 relative to those in class 0. Cells with $p - values$ higher than 0.05 are highlighted in gray, indicating that the difference in the distribution of the features between the two classes is not statistically significant.

Feature	**Caller**			**Callee**		
	GCC	**Clang**	**Selected**	**GCC**	**Clang**	**Selected**
FuncSize	△ 21.59%	△ 12.62%	✓	▽ 62.01%	▽ 68.00%	✓
AvgCallDist	△ 10.12%	△ 2.96%	✓	▽ 94.12%	▽ 95.01%	✓
LoopCount	△ 25.00%	▽ 25.00%	✗	△ 84.61%	▽ 56.45%	✗
SizeInc	△ 84.16%	△ 68.82%	✗	▽ 95.74%	▽ 96.11%	✓
BBInc	△ 77.10%	△ 65.42%	✗	▽ 95.82%	▽ 96.42%	✓
SizeZScore	△ 23.67%	△ 24.79%	✓	▽ 425.84%	▽ 340.03%	✓
BBZScore	△ 22.11%	△ 23.33%	✓	▽ 409.99%	▽ 362.62%	✓
InCallsZScore	△ 12.63%	△ 34.01%	✗	▽ 97.34%	▽ 96.74%	✓
OutCallsZScore	△ 46.62%	△ 3.50%	✗	▽ 352.77%	▽ 329.09%	✓
IsRecursive	△ 23.51%	△ 24.79%	✓	▽ 95.97%	▽ 96.95%	✓
o_reg	△ 17.58%	△ 16.81%	✓	▽ 41.24%	▽ 55.74%	✓
o_mem	△ 20.21%	△ 23.10%	✓	▽ 36.07%	▽ 59.19%	✓
o_phrase	△ 21.02%	△ 42.27%	✓	▽ 47.09%	▽ 63.22%	✓
o_displ	△ 8.04%	△ 4.39%	✗	▽ 22.41%	▽ 34.66%	✓
o_imm	△ 4.92%	△ 24.81%	✓	▽ 48.24%	▽ 53.94%	✓
o_near	△ 13.90%	△ 12.43%	✓	▽ 56.09%	▽ 63.40%	✓
o_fpreg	△ 609.27%	△ 801.86%	✓	△ 667.05%	△ 103.11%	✓

in class 0. Features are categorized based on their significance and direction of change, with distinctions made between caller and callee contexts across GCC and Clang.

Gray-shaded cells in Table 2 indicate features where the p-value of the

Wilcoxon test is greater than 0.05, meaning that the observed differences are not statistically significant. Such features are considered uninformative for distinguishing between the two classes and were excluded from further analysis. For all features, the percentage increases ($\triangle$) are highlighted in green, while percentage decreases ($\triangledown$) are shown in red to illustrate the direction of change between the two classes, regardless of statistical significance.

Features marked with a checkmark (✓) in the "Selected" column were included in the final feature set, as their differences were statistically significant for both compilers and indicative of meaningful patterns between the two classes. In contrast, features marked with a cross (✗) were excluded due to lack of significance. In our observations, we realized that including these features often improved the F1 score for one compiler while degrading it for the other. This imbalance in performance highlights the compiler-specific nature of these features, which could lead to biased models that fail to generalize effectively across different compilers.

By selecting only features that are statistically significant across both compilers, we intend to ensure that the final feature set emphasizes generalizable patterns rather than compiler-specific artifacts. Future work may revisit these excluded features to investigate their potential in compiler-aware models or scenarios targeting a single compiler family.

The results presented in Table 2 strongly align with our initial hypotheses regarding the factors influencing compiler inlining decisions. Specifically, features such as *FuncSize*, *SizeInc*, and *BBInc* align with our expectations that compilers balance the trade-off between reducing call overhead and avoiding excessive code expansion. The observed positive mean changes for FuncSize in the caller context and the significant negative changes in the callee context align with our assumption that smaller functions are generally favored for inlining, whereas larger functions are avoided to prevent unnecessary code bloat.

Moreover, the use of Z-scores, particularly *SizeZScore* and *BBZScore*, supports our expectation that compilers use relative metrics to assess a function's suitability for inlining. The extremely negative values observed in the callee context for these features show that functions significantly smaller and simpler than the program's average are strong candidates for inlining.

Finally, the results for operand-type features and *AvgCallDist* further support our assumptions. Especially, the negative mean changes for *AvgCallDist* of class 1 samples relative to class 0 align with our expectation that functions with shorter average distances from their callers are more likely to

be inlined. This negative change shows that, on average, functions in class 1 are closer to their callers compared to those in class 0.

Table 3: Distribution of inlining decisions for caller-callee pairs in our dataset. Positive (Pos.) classes refer to caller-callee pairs where the callee was inlined at both O2 and O3 optimization levels. Negative (Neg.) classes indicate cases where the callee was not inlined at one or both optimization levels. The table presents a breakdown of positive and negative cases for both GCC and Clang compilers. Notably, the positives at O2 and O3 can indicate situations where a function is inlined at certain call sites of the caller, while other call sites remain uninlined, or scenarios where the callee may be recursive.

Set	**Compiler**	**O0**		**O1**		**O2**		**O3**		**Total**
		Pos.	**Neg.**	**Pos.**	**Neg.**	**Pos.**	**Neg.**	**Pos.**	**Neg.**	
Train	GCC	19,112	103,083	6,500	84,658	649	79,037	451	75,687	369,177
	Clang	23,909	101,836	2,028	82,244	1,533	79,742	1,485	79,451	372,228
Test	GCC	9,537	54,322	3,645	46,819	372	43,083	300	41,507	199,585
	Clang	10,592	52,461	658	43,710	570	42,211	551	41,882	192,635
Total		**63,150**	**311,702**	**12,831**	**257,431**	**3,124**	**244,073**	**2,787**	**238,527**	**1,133,625**

3.3. Inlining Decision Prediction

The Random Forest (RF) classifier is an efficient non-linear ensemble model, where multiple decision trees are used to solve the same problem and improve overall performance. One of RF's key strengths is its ability to handle large datasets as well as its robustness to overfitting (Breiman, 2001). Additionally, RF is adept at classifying imbalanced data (Khoshgoftaar et al., 2007).

Table 3 illustrates the distribution of caller-callee pairs statuses within our dataset. A "Positive" status indicates that the callee is inlined at both O2 and O3 optimization levels. Conversely, a "Negative" status denotes callee that are either never inlined or inlined only at one of O2 or O3 levels. The data reveals a sever imbalance, with the "Positive" class being significantly rarer than the "Negative" class, highlighting a considerable skew in our dataset. Furthermore, the dataset employed in this research is both sizable and complex. Consequently, we opted to utilize an RF classifier for two primary reasons: it is adept at managing the imbalance present within our dataset and it effectively navigates the challenges posed by the dataset's size and complexity through its inherent non-linearity.

Given a set of caller-callee features $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$, the objective of our RF classifier is to predict the binary target variable $y \in \{0, 1\}$, where

$y = 1$ (Positive) if the callee should be inlined and $y = 0$ (Negative) otherwise. More specifically, the RF classifier aims to learn the mapping function $f : \mathbf{X} \rightarrow y$ such that the probability $P(y = 1 \mid \mathbf{X})$ is maximized for true Positive instances and minimized for true Negative instances.

To achieve this, each decision tree T_i within the forest makes an independent prediction $h_i(\mathbf{X}_s^i)$ based on a random subset of the features $\mathbf{X}$. The final prediction of the RF classifier is obtained through majority voting or averaging these individual tree predictions:

$$\hat{y} = \text{mode}\{h_1(\mathbf{X}_s^1), h_2(\mathbf{X}_s^2), \ldots, h_m(\mathbf{X}_s^m)\}$$

By averaging the probabilities from each tree, we obtain:

$$P(y = 1 \mid \mathbf{X}) = \frac{1}{m} \sum_{i=1}^{m} P_i(y = 1 \mid \mathbf{X}_s^i)$$

where m is the number of trees in the forest. We then utilize these predictions to adjust functions prior to feeding the language models.

The RF model was trained on our fixed-length feature vectors obtained by concatenating chosen features of callers and their callees. Specifically, each caller-callee pair was represented as a single row in the design matrix X, with its corresponding class label in y. To determine the optimal hyperparameters, we randomly partitioned the training data into an 80% subset for model fitting and a 20% subset for validation. We then performed a randomized search over a predefined range for each parameter, sampling combinations of $n\_estimators \in [50, 200]$, max_depth $\in [10, 30]$, max_features $\in [0.1, 1.0]$, min_samples_split $\in [2, 20]$, and criterion $\in$ {gini, entropy}, and evaluated performance on the validation set. The best configuration used 100 trees (n_estimators = 100), a maximum depth of 21, max_features = 0.6, min_samples_split = 10, and the "entropy" splitting criterion, with class_weight = balanced to mitigate class imbalance. Finally, we merged the training and validation sets and retrained the Random Forest on the full dataset to obtain the final model.

3.4. CFG Expansion

As discussed earlier, Oi represents a hypothetical optimization level at which we inline the subset $I_{Oi}(f)$ of functions that are commonly inlined at both O2 and O3. To transform functions from their initial optimization level

Algorithm 1 CFG Expansion

Require: f - target function
Ensure: Expanded CFG of f with eligible callees
1: $Q \leftarrow$ empty queue
2: Enqueue $(f, 0)$ into Q
3: **while** $Q \neq \emptyset$ **do**
4: $(f_{curr}, d_{curr}) \leftarrow$ Dequeue from Q
5: **for** each $c \in$ GetCallees(f_{curr}) **do**
6: **if** $c \neq f$ AND ShouldInline(c) **then**
7: **if** $\neg$InlinedAtLowerDepth(c, d_{curr}) **then**
8: $f \leftarrow$ ExpandCFG(f, c)
9: Enqueue $(c, d_{curr} + 1)$ into Q
10: **end if**
11: **end if**
12: **end for**
13: **end while**
14: **return** Modified CFG of f

to Oi, we trained an RF model to predict $I_{Oi}(f)$. Once we have this set of predicted inline targets, we apply a CFG expansion procedure to incorporate these callees into the target CFG, thereby constructing f_{Oi}.

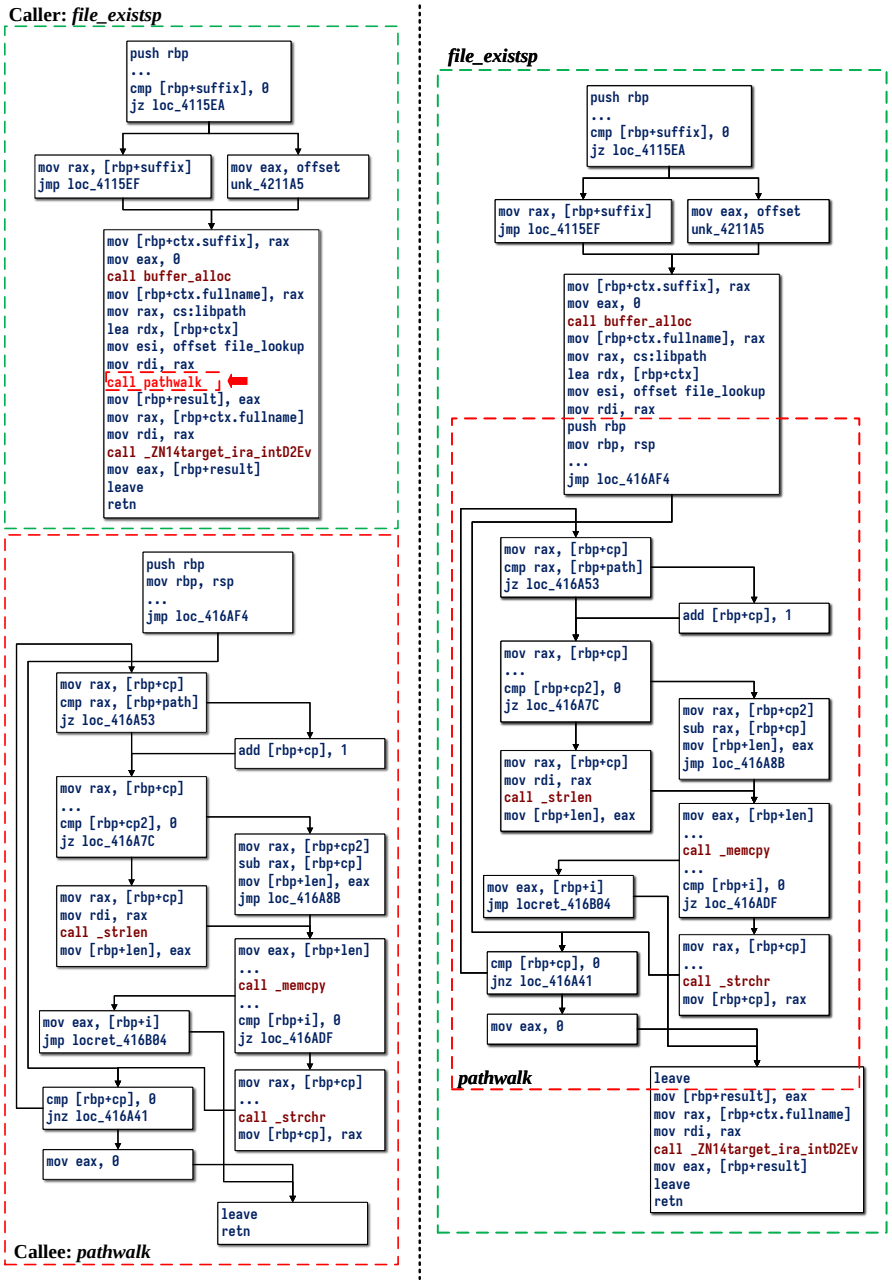
Our CFG expansion algorithm (Algorithm 1) is designed to systematically inline the predicted set $I_{Oi}(f)$ into f. Its key operations are as follows:

3.4.1. ***Initialization***

Starting with the baseline CFG of the target function f, we identify all direct call sites. Each callee is assigned a depth level (initially zero) and placed into a queue for possible inlining. The depth concept allows the algorithm to maintain a strict order of expansions, ensuring that it avoids inlining the same callee recursively through direct (recursive function) or indirect calls (called by another callee).

3.4.2. ***Breadth-First Search (BFS)***

The algorithm employs a BFS strategy to traverse the call hierarchy originating from f. When a callee is dequeued, the algorithm checks if it should be inlined based on the model's prediction (Line 6). If eligible and was not inlined at a lower depth (Line 7), it is merged into f at the given call site.



a) Original functions (Caller-callee pair) **b) Expanded function: *file_existsp***

Figure 4: Illustration of CFG Expansion Process.

Any new callees introduced by this inlining step are then enqueued, assigned an incremented depth, and considered in a subsequent iteration.

3.4.3. ***Expanding CFG***

The CFG expansion process (Line 8) transforms a caller function by merging its CFG with the CFG of a selected callee. The primary goal of this step is not to generate a fully runnable binary, but rather to create a richer representation for training language models. Thus, the exact register naming is less critical than the flow of logic and control. The representation learning model, trained on such CFGs, is expected to learn the underlying semantics of the code.

Fig. 4 shows an example of our CFG expansion strategy. In the illustrated example, the `file_existsp` function (caller) invokes the `pathwalk` function (callee) at a specific call site. To inline this call and expand the CFG of `file_existsp`, the call instruction is conceptually replaced with the body of `pathwalk`. During this process, the `retn` instruction in the callee's code is removed, as returning is no longer necessary once its instructions are directly integrated into the caller's control flow. The call site in `file_existsp` thus becomes the insertion point for the callee's CFG, integrating the two functions into a unified and continuous CFG.

4. Empirical Study

In our empirical study, we seek to address the following research questions:

RQ1: *What is the impact of function inlining on BCSD?*

RQ2: *Can compiler behavior regarding function inlining decisions be reasonably predicted after compilation?*

RQ3: *How does FIN affect the performance of BCSD?*

RQ4: *What is the computational overhead introduced by our proposed preprocessing technique in a BCSD pipeline?*

RQ5: *How does FIN impact the effectiveness of identifying real-world known vulnerabilities?*

Q1 is designed to explore the impact of frequently inlined functions on the BCSD. Q2 assesses the feasibility and effectiveness of our proposed method for predicting compiler inlining decisions. Q3 focuses on evaluating the effect of FIN on state-of-the-art binary code representation learning techniques. Q4 examines the efficiency of our proposed preprocessing technique within the BCSD pipeline. Lastly, Q5 investigates how FIN impacts the effectiveness of BCSD in identifying real-world known vulnerabilities.

4.1. Experimental Environment

In this study, all experiments were conducted on a server equipped with a 32-core AMD Ryzen Pro 3975WX CPU operating at 3.50 GHz, 500 GB of memory, 4 Nvidia RTX A6000 GPUs, and running Windows Server 2022 Datacenter. For disassembly purposes, we utilized IDA Pro version 8.0 (Hex-Rays, 2024a).

4.1.1. ***Baseline Models***

For the embedding generation and binary code clone search, we employed three state-of-the-art transformer models, CLAP Wang et al. (2024), jTrans (Wang et al., 2022), and Trex (Pei et al., 2020). Our choice was guided by two main considerations. Firstly, we prioritized models with publicly available official implementations, ensuring the validity and reliability of our results. Secondly, transformer-based models have demonstrated superior performance over other static BCSD techniques (Wang et al., 2022; Pei et al., 2020). For our experiments, we initially downloaded the pretrained jTrans[2] and Trex[3] models. These models were then fine-tuned using two versions of the BinKit dataset, details of which are provided subsequently. Throughout the fine-tuning process, we maintained the default hyperparameters as specified in the original configurations of these models. To ensure fairness in the fine-tuning process, we standardized the initial conditions for both dataset versions, including the training samples and the configurations of anchor, positive, and negative pairs. This approach ensures that any observed differences in model performance are attributable to the dataset variations and not to changes in the experimental setup. For CLAP Wang et al. (2024), however, we opted for a zero-shot evaluation rather than fine-tuning. This decision was motivated by two factors: first, CLAP's zero-shot performance

[2] `https://github.com/vul337/jTrans`
[3] `https://github.com/CUMLSec/trex`

Table 4: Number of functions and unique caller-callee pairs in the dataset used in this research.

	Set	GCC				Clang				Total
		O0	O1	O2	O3	O0	O1	O2	O3	
Functions	Train	62,016	46,489	44,666	42,953	61,300	43,255	43,236	43,122	387,037
	Test	36,194	26,240	27,064	26,922	35,141	23,924	23,901	23,687	223,073
Caller-callee pairs	Train	122,195	91,158	79,686	76,138	125,745	84,272	81,275	80,936	741,405
	Test	63,859	50,464	43,455	41,807	63,053	44,368	42,781	42,433	392,220

on BinKit was already reasonably high; second, using a zero-shot setting eliminated training effects, allowing us to isolate and assess the impact of CFG expansion alone during testing.

4.1.2. ***Datasets***

The dataset used in this research is a subset of BinKit (Kim et al., 2022), consisting of all functions compiled with Clang 13.0 and GCC 11.2.0 across four optimization levels (O0, O1, O2, and O3) for the x86-64 architecture. The dataset originally includes a total of 920,761 functions. However, for our experiments, we only selected the functions that originated from the source code and excluded any compiler-generated functions. This selection left us with a total of 610,110 functions. We then extracted unique caller-callee pairs from these functions.

To ensure meaningful evaluation and prevent information leakage, our train-test split was performed at the project level rather than at the function level. This means that entire projects (along with their associated functions) were placed exclusively in either the training or test set. Given the substantial variance in the number of functions per project, we first grouped projects by their size. Subsequently, we randomly selected a set of projects to achieve a roughly 70–60% training and 30–40% testing proportion. This strategy was employed to maintain a representative distribution of functions while ensuring no overlap of project-related information between the training and test sets.

Table 4 provides a detailed breakdown of the number of functions and caller-callee pairs for each compiler and optimization level in both the training and test sets. The BinKit subset was primarily used for the BCSD task, while the caller-callee pairs were utilized for function inlining prediction and analysis.

More specifically, we leveraged the caller-callee dataset to generate inlining ground-truth and to train FIN. The generated ground-truth was then used to preprocess BinKit functions in the training set, while FIN predictions were used to preprocess functions in the test set. We subsequently fine-tuned and tested Trex and jTrans on both the original BinKit and the preprocessed BinKit datasets.

4.1.3. ***Evaluation metric***

To evaluate the performance of FIN in predicting compiler behavior regarding inlining decisions, we use precision, recall, F1 score, and AUC. Furthermore, we use the Mean Reciprocal Rank (MRR) to measure and compare the performance of the baseline models with and without FIN preprocessing. MRR is a statistic used to evaluate the performance of a system that produces a list of possible responses to a query, ranked by their relevance. MRR is defined as the average of the reciprocal ranks of the first relevant response for a set of queries.

Let Q be the set of queries, and let $|Q|$ denotes the number of queries in Q. For a given query $q \in Q$, let rank_q denotes the rank position of the first relevant response in the list of results produced by the system for query q.

The reciprocal rank for query q is then given by:

$$\text{ReciprocalRank}(q) = \frac{1}{\text{rank}_q}$$

The Mean Reciprocal Rank (MRR) is the mean of the reciprocal ranks for all queries in the set Q:

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{rank}_q}$$

The MRR provides a single-figure measure of quality across multiple queries, with higher MRR values indicating better performance. It effectively captures the ability of the system to rank relevant results higher in the response list.

4.1.4. ***Clone search task***

Clone search is a BCSD task where the goal is to identify similar or identical code fragments across different binary executables. Let $Q = \{q_1, q_2, \ldots, q_n\}$ represents a set of binary functions in the query set and $P = \{p_1, p_2, \ldots, p_m\}$

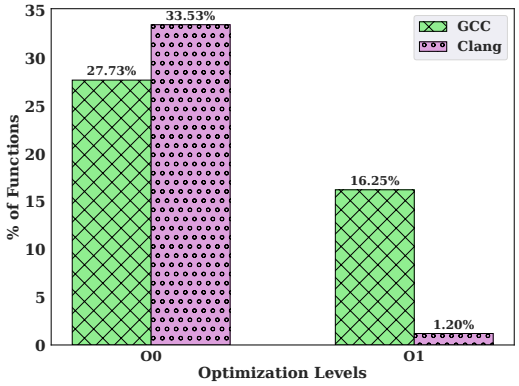


Figure 5: Comparison of the number of functions identified for CFG expansion at optimization levels O0 and O1 relative to the total number of functions in our dataset.

represents a pool of binary functions. The objective of the clone search task is to identify functions in P that are similar or identical to each function in Q.

For each function $q_i \in Q$, the task is to find the function(s) $p_j \in P$ that maximize a similarity measure $S(q_i, p_j)$. Techniques such as graph matching, embedding models, or machine learning classifiers are employed to compute $S(q_i, p_j)$.

Formally, for each $q_i \in Q$, we seek to find the function $p_j \in P$ that maximizes $S(q_i, p_j)$, i.e.,

$$\forall q_i \in Q, \text{find}\, p_j \in P \,\text{such that}\, S(q_i, p_j) = \max_{p_k \in P} S(q_i, p_k)$$

The similarity scores $S(q_i, p_j)$ are then sorted in descending order for each q_i, allowing the identification of the most similar functions $p_j \in P$ to each query function q_i.

Following the literature (Hu et al., 2018; Marcelli et al., 2022; Wang and Wu, 2017; Xu et al., 2023), we use a pool size of 500 query functions from O_m and 500 corresponding functions from O_n in the repository for our clone search experiments, where m and n are in the range [0-3] and $m \neq n$. To ensure robustness and reliability of our results, we repeat the test 100 times, each time selecting 500 random query functions and their corresponding functions. After conducting these repeated tests, we employ the Wilcoxon signed-rank test to statistically compare the results.

4.2. RQ1: Function Inlining Impact

To answer RQ1, we conducted three experiments on the test functions directly without applying FIN. We first estimated the portion of functions affected by function inlining. For this purpose, after generating the ground-truth inlining, we collected functions at the O0 and O1 optimization levels that have at least one callee identified as inlined at both the O2 and O3 optimization levels. As illustrated in Fig. 5, our observations show that at optimization level O0, 27.73% of the functions in GCC and 33.53% of the functions in Clang contain callees that are inlined at both O2 and O3. Similarly, at optimization level O1, 16.25% of the functions in GCC and 1.20% of the functions in Clang exhibit such inlining behavior.

Next, we measured the similarities of function pairs using embeddings obtained from jTrans and Trex, which were fine-tuned on the BinKit dataset. As shown in Fig. 6, the highest similarity drop occurs when comparing functions at O0 with those at O2 or O0 with those at O3. We observed that the presence of function inlining can cause a similarity drop of up to 5.58%, 8.96%, and 26.78% for CLAP, jTrans, and Trex, respectively. Additionally, we noticed that jTrans is more robust against function inlining than Trex.

To evaluate the effect of similarity drops in binary clone search, we designed an experiment where we separated functions with inlining from those without inlining. We then queried functions at O0 against a pool of O3 functions for both groups. Fig. 7 shows the average MRR values obtained from 100 repetitions for each experiment. We observed that the MRR values drop by up to 21.84%, 18.42%, and 49.01% when performing clone search on functions affected by inlining using CLAP, jTrans, and Trex, respectively.

The observed drop in MRR suggests three possible reasons:

1. **Inherent Function Complexity:** Functions that do not have inlinable call sites might be inherently simpler than those that do. This simplicity could complicate BCSD. However, since language models such as jTrans and CLAP are designed to understand the semantics of binary functions, this is unlikely to be the primary reason for the drop in MRR.
2. **Semantic Enrichment through Function Inlining:** When comparing functions at O0 with those at higher optimization levels, function inlining introduces the 1-to-n matching problem, significantly complicating BCSD. Function inlining adds the semantic of the inlined callee to the caller function, enriching the O3 versions with additional

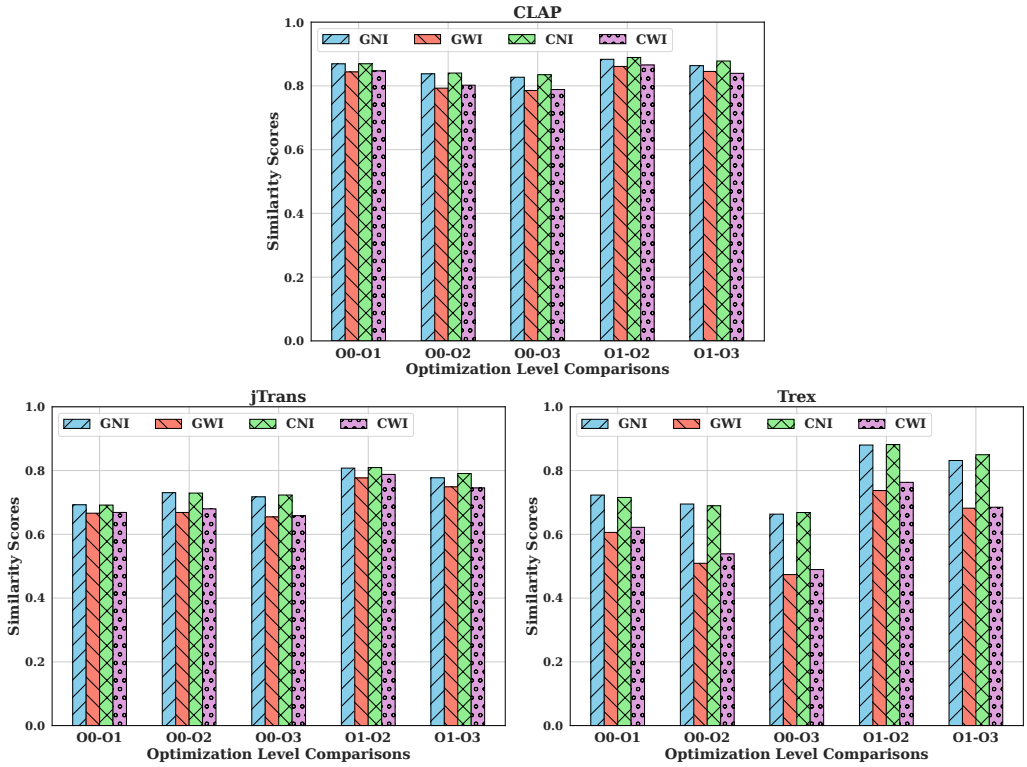


Figure 6: Comparison of similarity scores by compiler and inlining settings. In this figure, GNI refers to GCC-compiled functions without inlining, and GWI to those with inlined call sites. Similarly, CNI and CWI refer to Clang-compiled functions without and with inlining, respectively.

context and information. This enrichment causes the similarity between O0 and O3 versions of the same function to drop (see Fig. 6), as the added semantics alter the structure and behavior of the function.

3. **Post-Inlining Optimizations:** Post-inlining optimizations can further alter functions and complicate BCSD. These transformations often depend on the initial inlining step, making function inlining the starting point for subsequent structural and semantic changes in the code.

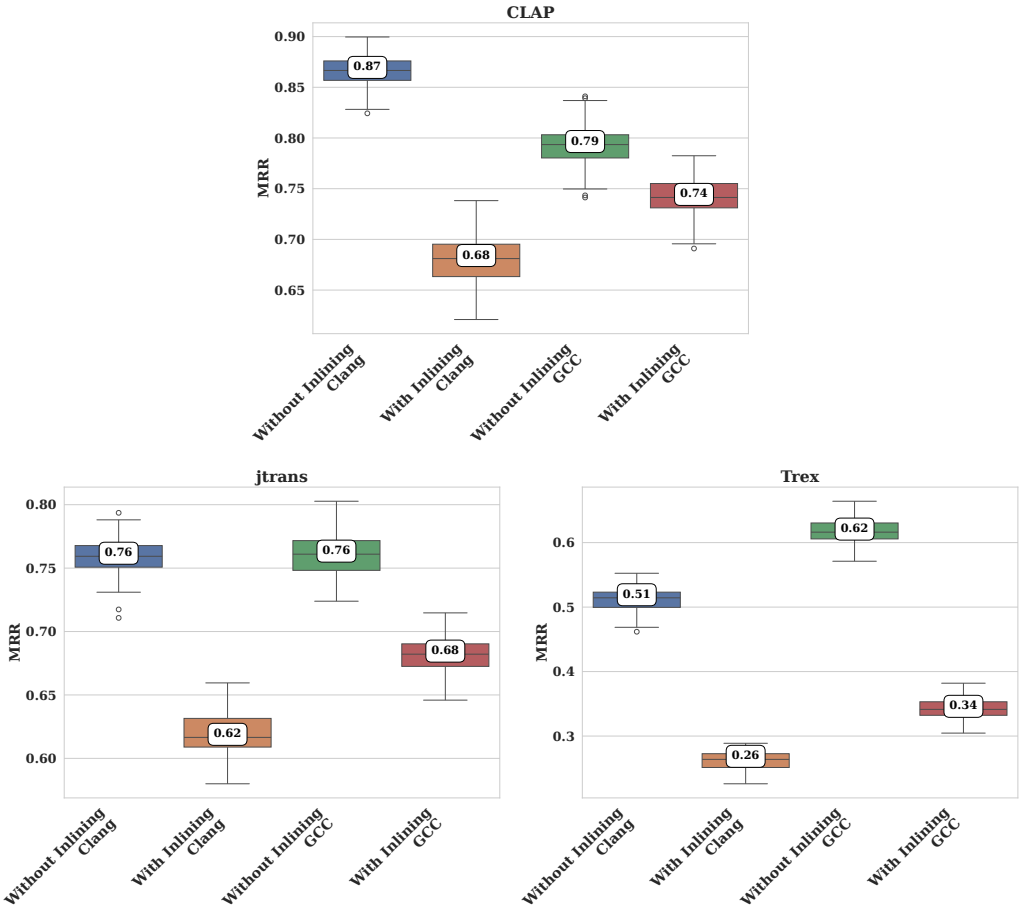


Figure 7: Comparative analysis of MRR for functions compiled with optimization level O0, searched within a repository of functions compiled at optimization level O3, with and without function inlining.

Answer to RQ1: Our observations show that functions containing inlined callees correlate with lower similarity scores and reduced retrieval performance in BCSD tasks. The substantial difference in inlining between low and high optimization levels is associated with a notable performance drop in state-of-the-art BCSD approaches. Since function inlining often serves as a starting point for further optimizations, its presence may contribute to more complex transformations that complicate similarity detection. While these findings suggest that inlining and the subsequent optimizations may influence the observed difficulties, further investigation is needed to establish a direct causal relationship.

Table 5: Performance comparison of FIN, BinGo, Asm2Vec, and OpTrans in predicting whether a callee should be inlined.

Method	Compiler	Precision	Recall	F1 Score	AUC
BinGo (Chandramohan et al., 2016)	GCC	0.36	0.25	0.30	N/A
	Clang	0.45	0.23	0.30	N/A
Asm2Vec (Ding et al., 2019)	GCC	0.41	0.24	0.30	N/A
	Clang	0.44	0.18	0.26	N/A
OpTrans (Sha et al., 2025)	GCC	0.23	0.21	0.22	N/A
	Clang	0.23	0.16	0.19	N/A
FIN	GCC	**0.75**	**0.61**	**0.67**	**0.91**
	Clang	**0.85**	**0.72**	**0.78**	**0.95**

AUC values are not available for the BinGo, Asm2Vec, and OpTrans rule-based approaches.

4.3. RQ2: Compiler Inlining Decision Predictions

We trained an RF model using the proposed features to predict the inlining decisions made by compilers and evaluated its performance on our test set. Additionally, we implemented the BinGo (Chandramohan et al., 2016), OpTrans (Sha et al., 2025), and Asm2Vec (Ding et al., 2019) strategies for CFG expansion to compare with our approach, FIN. Table 5 presents the results obtained by the four approaches. Since BinGo, Asm2Vec, and OpTrans are rule-based approaches and do not provide probabilities, we could not calculate the AUC value for them. The results show that while our counterparts could not achieve an F1 score higher than 0.30, FIN achieved an average F1 score of 0.72, which is 140% higher than that of BinGo and Asm2Vec. This shows that relying solely on the in and out degree of the callee function (as introduced by BinGo) and function sizes (as added by Asm2Vec and also used in OpTrans) might not be sufficient for making CFG expansion decisions. To further investigate this, we analyzed the contribution of our selected features by obtaining importance of features from our RF model.

In RF, feature importance is quantified using the Mean Decrease in Impurity (MDI). Let S be the set of training samples, and let $H(S)$ represents the entropy of S. For a given feature A, the information gain $IG(S, A)$ from splitting S on A is calculated as:

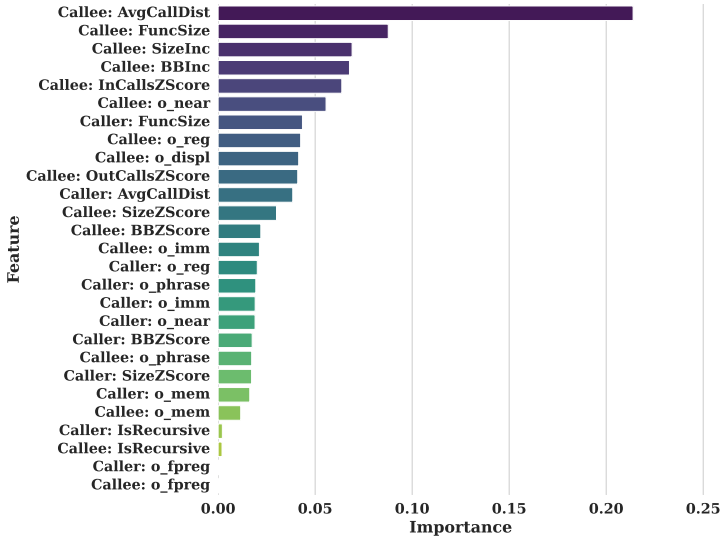


Figure 8: The importance values obtained from the RF model.

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$$

where S_v is the subset of S for which feature A takes the value v, and Values(A) represents all possible values of A. The feature importance for A is then determined by summing the information gain $IG(S, A)$ over all nodes, where A is used for splitting across all trees in the forest, and then averaging these sums over all trees. Formally, for T trees in the forest and node t in tree T_i:

$$\text{Importance}(A) = \frac{1}{T} \sum_{i=1}^{T} \sum_{t \in \text{nodes}} IG_t(S, A)$$

This aggregated measure represents the total reduction in entropy attributable to feature A throughout the forest.

Fig. 8 illustrates the contribution of each feature in making correct decisions about whether a callee function should be inlined. The results reveal that, contrary to the common belief that function size and the in and out degrees of a callee function are the most important factors for deciding CFG expansion, the average distance of a callee function from its callers is actually the most significant feature.

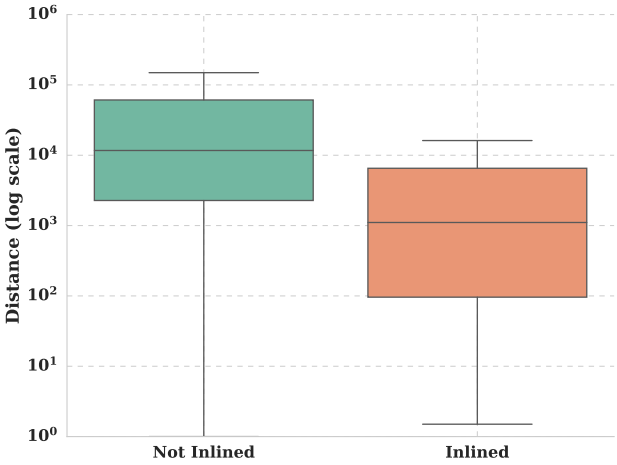


Figure 9: Boxplot of average distance of callees from their callers in both class 0 (Not Inlined) and class 1 (Inlined).

Upon further examination of the average incoming call distances of callees, we found that functions that are very far from their callers are less likely to be suitable choices for CFG expansion. As shown in Table 2, the average distance of samples in class 1 is 94.14% and 95.01%, lower than those in class 0 for GCC and Clang, respectively. Furthermore, upon analyzing all callees in our dataset, we discovered that 40% of the functions in class 0 had an average distance greater than the maximum average distance observed in class 1 (Fig. 9).

> *Answer to RQ2: Our findings show that although GCC and Clang compilers make inlining decisions based on different heuristics, which can also vary by program, the overall structure of a binary executable and certain function features can help predict a considerable number of the inlining decisions made by these compilers.*

4.4. RQ3: Effectiveness of FIN

To evaluate the effect of FIN predictions, we conducted an experiment using the BinKit dataset with three CFG expansion techniques: BinGo, Asm2Vec, OpTrans, and FIN. We preprocessed the dataset with each of these techniques, collected functions that were affected by inlining according to our ground-truth data, as well as extracted their embeddings from

Table 6: Comparison of the MRR values for clone searches on functions that include at least one inlined callee, using different CFG expansion strategies relative to using no strategy. Cells with p-values higher than 0.05 are highlighted in gray, indicating that the improvement is not statistically significant.

Model	Compiler	Query - Pool	None	BinGo		Asm2Vec		OpTrans		FIN	
			MRR	MRR	Impr.	MRR	Impr.	MRR	Impr.	MRR	Impr.
jTrans	GCC	**O0 - O1**	0.77	0.79	1.69%	0.78	1.02%	0.79	1.76%	0.80	**3.44%**
		O0 - O2	0.74	0.75	1.97%	0.74	0.87%	0.74	0.41%	0.76	**3.57%**
		O0 - O3	0.71	0.72	1.76%	0.72	0.66%	0.71	-0.44%	0.73	**2.77%**
	Clang	**O0 - O1**	0.66	0.66	0.32%	0.65	-1.32%	0.66	-0.02%	0.69	**5.57%**
		O0 - O2	0.65	0.64	0.54%	0.64	-1.55%	0.65	-0.11%	0.68	**5.40%**
		O0 - O3	0.64	0.64	0.33%	0.63	-1.58%	0.64	-0.15%	0.67	**5.08%**
Trex	GCC	**O0 - O1**	0.55	0.56	1.39%	0.55	0.21%	0.57	3.60%	0.58	**4.79%**
		O0 - O2	0.41	0.43	4.99%	0.43	3.93%	0.41	-0.40%	0.45	**9.62%**
		O0 - O3	0.36	0.39	10.50%	0.39	9.02%	0.37	2.99%	0.40	**12.48%**
	Clang	**O0 - O1**	0.28	0.31	9.48%	0.30	7.71%	0.31	11.20%	0.34	**21.54%**
		O0 - O2	0.27	0.30	11.10%	0.29	9.01%	0.30	11.84%	0.32	**19.99%**
		O0 - O3	0.27	0.30	11.14%	0.29	9.10%	0.30	11.11%	0.32	**19.73%**
CLAP	GCC	**O0 - O1**	0.84	0.84	-0.18%	0.84	-0.30%	0.85	0.83%	0.87	**3.21%**
		O0 - O2	0.74	0.76	1.82%	0.76	1.93%	0.74	-0.18%	0.76	**2.66%**
		O0 - O3	0.75	0.75	**0.66%**	0.75	0.62%	0.73	-1.70%	0.75	-0.12%
	Clang	**O0 - O1**	0.67	0.69	2.77%	0.69	2.58%	0.68	1.07%	0.75	**11.01%**
		O0 - O2	0.68	0.70	2.58%	0.70	2.26%	0.68	0.84%	0.74	**8.05%**
		O0 - O3	0.68	0.70	2.32%	0.70	2.06%	0.68	0.82%	0.73	**7.64%**

Note: MRR values were originally calculated with 16-bit floating-point precision, and improvements were computed based on these 16-bit values. However, both MRR and improvement percentages are reported to two decimal.

both the original and preprocessed data compiled with GCC and Clang using CLAP, jTrans and Trex. This process resulted in a total of 30 sets of embeddings.

Next, we performed clone search experiments on these 30 sets. Specifically, for each set, we randomly selected a query set of O0 embeddings and a pool of randomly selected functions at O1, O2, and O3 optimization levels. The rationale behind this selection is that the BCSD between O0 and the other three optimization levels is significantly challenging, making these settings suitable for observing potential improvements.

In total, we designed 18 experiments, each incorporating a combination of the following variables: compiler (GCC or Clang), query/pool optimization

setting (e.g., O0/O1), CFG expansion technique (None, BinGo, Asm2Vec, OpTrans, and FIN), and representation learning technique (CLAP, jTrans, or Trex). For fair comparison, we used a single random set of functions in each experiment for each CFG expansion technique, with different sets for each repetition.

Table 6 presents the MRR values obtained from the clone search experiments conducted with functions affected by inlining. The improvements for each strategy were calculated relative to the MRR obtained from the no inlining strategy. The results show that, in the presence of function inlining, FIN improved the performance of jTrans, CLAP, and Trex by up to 5.57%, 11.01%, and 21.54%, respectively, outperforming the BinGo and Asm2Vec strategies.

The improvements in Table 6 were largely expected given how OpTrans, BinGo, and Asm2Vec handle function inlining. The three of them rely on selective inlining guided by a set of tuned thresholds, such as function size limits and coupling scores, that do not fully reflect how compilers actually decide which functions to inline at different optimization levels. Although these heuristics can prevent code-size explosion and maintain scalability, they often miss or misrepresent the detailed inlining behavior performed by the compiler. In contrast, FIN strives to predict and mirror the compiler's real inlining decisions rather than relying on inflexible, handcrafted rules. As a result, FIN covers a wider range of inlining scenarios and more accurately represents the inlined code, leading to the performance gains observed in Table 6.

To evaluate how false positives from FIN affect clone search when dealing with functions that were not originally affected by inlining, we repeated a similar set of experiments, this time employing only FIN. Specifically, we first collected functions that were not affected by inlining. Then, we designed 18 additional experiments for these collected functions. We maintained the same compiler and optimization settings and the same representation learning techniques as in the previous experiments. However, we excluded OpTrans, BinGo and Asm2Vec from this part, due to their lesser relevance and space limitations.

Table 7 shows the results obtained from the clone search experiments conducted with functions that were not originally affected by function inlining. The results indicate that in 77% of the experiments, the difference is negligible. However, FIN achieved notable improvements in some of experiments, suggesting two hypotheses for future study. The first hypothesis

Table 7: Comparison of the MRR values for clone searches on functions that include no inlined callee, using different CFG expansion strategies relative to using no strategy. Cells with p-values higher than 0.05 are highlighted in gray, indicating that the improvement is not statistically significant.

Compiler	Query - Pool	jTrans			Trex			CLAP		
		None	FIN		None	FIN		None	FIN	
		MRR	MRR	Impr.	MRR	MRR	Impr.	MRR	MRR	Impr.
GCC	**O0 - O1**	0.78	0.78	**-0.18%**	0.71	0.71	**-0.16%**	0.90	0.90	**-0.40%**
	O0 - O2	0.78	0.79	**0.80%**	0.66	0.65	**-1.28%**	0.81	0.81	**-0.15%**
	O0 - O3	0.76	0.76	**0.74%**	0.62	0.62	**0.33%**	0.79	0.79	**-0.10%**
Clang	**O0 - O1**	0.75	0.76	**0.45%**	0.52	0.55	**4.96%**	0.86	0.86	**-0.25%**
	O0 - O2	0.76	0.77	**0.65%**	0.51	0.54	**4.84%**	0.87	0.87	**-0.03%**
	O0 - O3	0.76	0.76	**0.01%**	0.51	0.54	**5.28%**	0.87	0.87	**-0.01%**

Note: MRR values were originally calculated with 16-bit floating-point precision, and improvements were computed based on these 16-bit values. However, both MRR and improvement percentages are reported to two decimal.

is that having functions with expanded CFGs during training or fine-tuning may make it easier for the language model to learn. The second hypothesis suggests that even though some functions are falsely inlined, they might still add more context and semantic information, making the functions more distinguishable.

By specifically targeting the challenges posed by function inlining, FIN demonstrates a more pronounced improvement in scenarios where models are highly impacted by inlined code. Our experimental results suggest that Trex, which experiences a greater drop in MRR when handling inlined functions, benefits substantially from the mitigation provided by FIN. In contrast, CLAP and jTrans, having been pretrained on a broader set of functions, including those compiled at various optimization levels, exhibit better baseline resilience to the effects of inlined code. Consequently, the improvements observed with FIN are more modest for CLAP and jTrans, as their performance is less affected by inlining-related discrepancies. Furthermore, the gains in CLAP's performance show that FIN can potentially enhance representation-learning models even without additional fine-tuning.

These findings suggest that the enhancements achieved by FIN do not simply inflate performance metrics by exploiting weaknesses in models such as Trex, but rather highlight FIN's ability to effectively address inlining-

specific challenges. This is particularly observed in cases where baseline models struggle with inlined code, emphasizing FIN's role in targeted mitigation rather than artificially boosting results.

> *Answer to RQ3: The results of our experiments show that FIN can significantly boost the performance of state-of-the-art assembly representation learning techniques. This represents a step forward in achieving accurate and robust cross-optimization BCSD.*

4.5. RQ4: Efficiency Analysis

FIN consists of three main steps: feature extraction, callee inlining prediction, and CFG expansion. The feature extraction step requires FIN to analyze the entire binary program. This process involves disassembling the binary, which inherently requires visiting each instruction in the program. As a result, the time complexity of feature extraction is $O(n)$, where n is the number of instructions in the program. However, since disassembly is a prerequisite for most BCSD pipelines, the feature extraction step in FIN can be integrated into this process, effectively leveraging the same operation without introducing additional overhead to the overall pipeline.

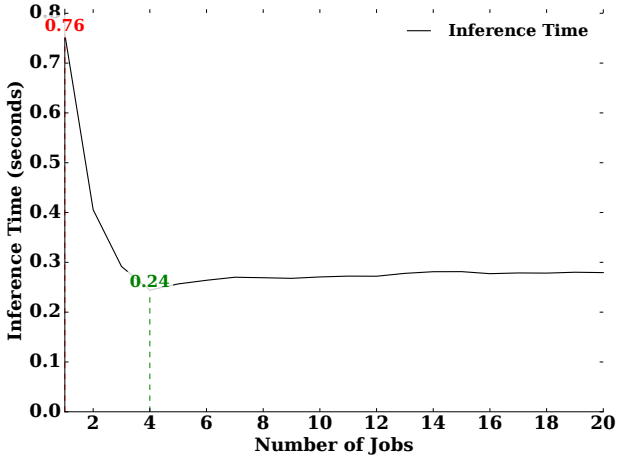


Figure 10: Average inference time (in seconds) for predicting the inlining status of 100,000 caller-callee pairs using different numbers of concurrent workers.

To evaluate the efficiency of the RF model used in FIN, we predicted the inlining status of 100,000 caller-callee pairs using 1 to 20 concurrent

workers to determine the best and worst inference times on our machine. We repeated each experiment 100 times and measured the prediction time. Fig. 10 shows the average prediction time for different numbers of concurrent workers. The experiments were conducted in a WSL Ubuntu environment with resources limited to 16GB of RAM and 4 CPU cores, along with reduced CPU scheduling priority to simulate a resource-constrained environment. In the worst case, it took 0.76 seconds with a single job to predict the labels of 100,000 caller-callee pairs, while in the best case, it took 0.24 seconds using 4 concurrent workers.

The time complexity of our CFG expansion for a single function is $O(c^d)$, where c denotes the number of call sites associated with caller-callee pairs identified for inlining by our prediction model, and d represents the depth of the call tree for the selected inlined calls. These caller-callee pairs are extracted during the feature extraction phase as part of the disassembly process. As a result, there is no additional overhead introduced for identifying call sites and caller-callee pairs beyond what is required for disassembly. In practice, functions typically invoke a limited number of other functions (resulting in a small c), and the call depth tends to be relatively shallow (resulting in a small d). This ensures that the CFG expansion remains computationally manageable within the overall BCSD pipeline.

> *Answer to RQ4: Overall, our analysis shows that FIN introduces low relative overhead to the BCSD pipeline. Its feature extraction is performed in conjunction with the required disassembly step, and the analysis of inlining prediction by a Random Forest model shows a manageable overhead, making it practical for integration into the BCSD pipeline.*

4.6. RQ5: Real-World Application

BCSD plays a critical role in vulnerability search by allowing analysts to match known security flaws against potentially different versions or variants of compiled code. When developers reuse code or apply standard libraries across multiple platforms, these similarities can remain hidden, sometimes subtly, by compiler optimizations, function inlining, or other transformations. BCSD aims to abstract away low-level differences, focusing instead on the core program logic. By identifying similar structural and semantic features in binaries, BCSD approaches help security researchers find known

Table 8: CVEs identified for each program in the BinKit dataset.

Program	CVEs
a2ps-4.14	CVE-2015-8107
tar-1.34	CVE-2022-48303
sharutils-4.15.2	CVE-2018-1000097
cpio-2.12	CVE-2010-4226, CVE-2019-14866, CVE-2021-38185
cflow-1.7	CVE-2023-2789
patch-2.7.5	CVE-2016-10713, CVE-2018-6951, CVE-2018-6952, CVE-2019-13636, CVE-2018-20969, CVE-2019-20633
libmicrohttpd-0.9.75	CVE-2023-27371
binutils-2.40	CVE-2023-1972, CVE-2023-25586, CVE-2025-0840
inetutils-2.4	CVE-2023-40303

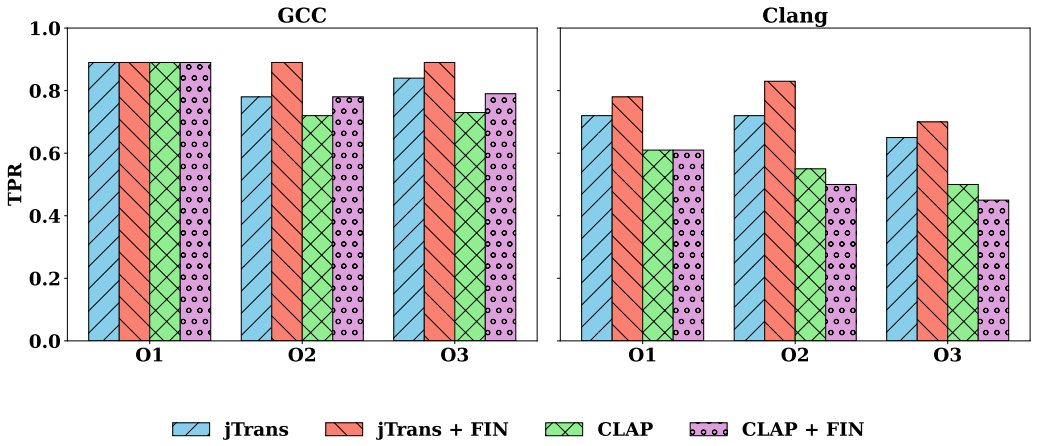


Figure 11: Average TPR for CVE retrieval using jTrans and CLAP, comparing performance before versus after FIN.

vulnerabilities in newly compiled or proprietary software.

To investigate the impact of FIN on the known vulnerability search task, we first extracted the CVEs associated with the programs in the BinKit dataset from the National Vulnerability Database. Next, we manually examined each CVE to establish a mapping between the CVEs and their corresponding source functions. We then leveraged assembly-to-source mappings to build ground-truth associations between assembly functions and CVEs[4]. In total, we collected 18 CVEs; however, we omitted four CVEs related to

[4]`https://github.com/McGill-DMaS/BinKit_CVE`

binutils and inutils because those programs had already appeared during our training phase. Table 8 summarizes the CVEs identified for each program.

We selected jTrans and CLAP, the two embedding models that performed best in our previous experiments, and evaluated them before and after applying FIN. For each model variant, we queried every vulnerable function compiled at O0 against the same program compiled at O1, O2, and O3 (evaluating each level separately). Finally, we computed the average true positive rate (TPR) for each model at each optimization level.

As illustrated in Figure 11, FIN has a generally positive impact on CVE retrieval. Except for CLAP on programs compiled with Clang at optimization levels O2 and O3, applying FIN resulted in equal or improved TPR. On average, jTrans + FIN achieved an 83% TPR compared to 76% for jTrans alone, and CLAP + FIN achieved a 67% TPR.

> *Answer to RQ5: Our results suggest that FIN can provide benefits for identifying real-world known vulnerabilities. In the majority of queries, adding FIN to jTrans and CLAP improved the TPR, suggesting that its inline-neutralizing approach can help mitigate some of the inconsistencies introduced by compiler optimizations.*

5. Discussion

The following sections discuss the strengths, the design choice, and shortcomings of our FIN approach, providing an analysis of its performance and applicability. We begin by outlining the primary limitations that hinder FIN from fully resolving the cross-optimization clone search problem. Subsequently, we examine the potential threats to the validity of our results, highlighting factors that can affect the reliability and generalizability of our findings. We then provide further justification regarding the selection of callee functions that were inlined at both O2 and O3. Finally, we explore the adaptability of FIN to different optimization levels, specifically addressing its effectiveness under the Os optimization setting. Through this structured examination, we aim to present a balanced view of FIN's capabilities and identify avenues for future enhancements.

5.1. Limitations

Although FIN improved the performance of the CLAP, jTrans and Trex models, the resulting MRR values indicate that the cross-optimization clone

search problem remains unsolved. CFG expansion alone may not be sufficient to address this challenge. Three potential issues justify this conclusion.

First, when a compiler inlines a function, it adjusts registers, memory references, and performs further optimizations on the caller function. FIN, on the other hand, merely replaces the call site with the assembly code of the callee function. Thus, even though the callee is added to the caller function, its representation may still differ from the compiler-inlined version, leading existing approaches to fail in capturing the similarity between the two.

Second, although CFG expansion is expected to add extra semantic value to the function, in practice, if the callee function is very similar to the caller function or if the callee function is too small, it might not add significant information, resulting in no performance improvement.

Third, Link-Time Optimization (LTO) poses additional challenges. LTO enables the compiler to perform optimizations, including aggressive cross-translation-unit function inlining, during the linking phase rather than at compile time. This process can significantly alter the call graph and function boundaries, complicating inlining detection and rendering our method less effective when applied to LTO-enabled binaries.

Our proposed FIN method operates within the boundaries of individual compilation units, relying on debug information extracted from binaries compiled without LTO enabled. As a result, the function inlining decisions observed in our dataset may not reflect the more aggressive or cross-unit inlining strategies applied in LTO-enabled compilations.

To address this limitation, future work could extend our approach to incorporate LTO-specific features by analyzing binaries compiled with LTO enabled and comparing their instruction-to-source mappings. Despite this limitation, we hypothesize that the core principles of our approach—capturing consistent inlining decisions at higher optimization levels—remain applicable, even under LTO, albeit with adjustments to account for its broader scope.

5.2. Threats to validity

There are three major threats to the validity of our proposed method that should be considered:

First, our approach relies heavily on IDA Pro for disassembly and feature extraction. Any inaccuracies or limitations inherent to IDA Pro could directly impact the accuracy and reliability of our results. For instance, if IDA Pro fails to correctly disassemble a binary or misidentifies certain features, the subsequent analysis and predictions made by FIN could be flawed.

Second, our ground truth generation process assumes that function mappings derived from debug information (using the *.debug_line* section) provide a reliable basis for identifying inlined functions. While prior research (Jia et al., 2023) supports this assumption, we acknowledge that debug information can be affected by compiler optimizations, resulting in partial or inconsistent mappings.

Lastly, we utilize the BinKit dataset for our experiments. Any issues or inaccuracies in the compilation process of the BinKit dataset could lead to invalid or biased results. If the dataset contains errors, such as incorrectly compiled binaries or mislabeled binary files, our method's performance metrics might not accurately reflect its true effectiveness.

5.3. Intersection of O2 and O3

In this study, we decided to predict and inline only those caller–callee pairs that were inlined at both O2 and O3 to strike a balance between coverage and consistency, covering approximately 90% of the functions inlined across all optimization levels. An alternative approach, taking the union of inlining relationships at O1, O2, and O3, would guarantee 100% ground-truth coverage; however, we preferred our intersection-based design choice for three main reasons. First, prior works comparing O1, O2, and O3 consistently reported high recall and MRR scores across those levels Ding et al. (2019); Pei et al. (2020); Wang et al. (2022, 2024), suggesting that the remaining 10% of inlining discrepancies have only a marginal impact on BCSD. Second, the union set includes many rarely inlined functions that introduce noise and complexity into the learning process. In our preliminary experiments, incorporating these rare cases resulted in a 12% drop in precision, leading to a substantial increase in false positives that can negatively impact functions that are never subject to inlining. Third, expanding the CFGs with too many callees, given the limited token budget of modern language models, risks pushing the primary caller function out of the model's input window, thereby undermining effective feature extraction.

5.4. Handling Os Optimization

The proposed FIN approach is primarily designed to address inlining decisions across optimization levels O0, O1, O2, and O3. However, it is important to discuss the potential applicability of FIN to the Os optimization level, which prioritizes reducing code size while maintaining performance.

The Os optimization level often applies similar inlining strategies as O2 and O3 but imposes additional constraints focused on minimizing binary size.

Preliminary observations suggest that functions inlined under -Os exhibit patterns comparable to those observed at O2 and O3. Functions inlined at both O2 and O3 are likely already inlined at -Os, and for those not inlined under Os, applying FIN will likely resolve inconsistencies. Conversely, functions inlined at Os but not at O2 or O3 are expected to have a minimal impact, similar to O1, making them less problematic for BCSD. These observations suggest that FIN can adapt to Os without substantial modifications. While our current evaluation does not explicitly cover Os, future work could include an empirical analysis of FIN's performance on binaries optimized at this level.

6. Related Work

BCSD is a critical area of research within computer network security, focusing on comparing binary files to identify similarities. BCSD has critical applications, including software vulnerability detection (Luo et al., 2023; Yu et al., 2021; Zhao et al., 2019) and malware analysis (Sun et al., 2023; Molloy et al., 2022; Li et al., 2021). The fundamental process of BCSD involves three main stages: code preprocessing, comparison unit generation, and similarity calculation. During code preprocessing, irrelevant instructions are removed and instructions are normalized to improve detection accuracy and efficiency (Xu et al., 2023; Guo et al., 2023). The comparison unit generation stage transforms binary code into an intermediate representation, such as byte streams or feature vectors (Sun et al., 2024; Gu et al., 2023). Finally, similarity computation is performed using methods such as vector distance calculation or subgraph matching (Shalev and Partush, 2018). BCSD faces several challenges, including variations due to compiler optimizations, differences across platforms, and code obfuscation techniques (Ding et al., 2019; Xue et al., 2019).

Representation learning techniques—particularly those that leverage language models—have gained prominence in BCSD, aiming to produce embeddings that capture semantic similarity despite low-level differences. For instance, Asm2Vec (Ding et al., 2019) employs the Paragraph Vector–Distributed Memory (PV-DM) model to embed instructions, while Trex (Pei et al., 2020) uses a transformer-based architecture to learn unified function representations across different platforms and compilers. jTrans (Wang et al.,

2022) and BinShot (Ahn et al., 2022) have applied BERT-based encoders with contrastive learning objectives to achieve cross-compiler function embeddings. More recently, CLAP (Wang et al., 2024) introduces Contrastive Language-Assembly Pre-training, aligning binary code with natural language explanations to learn transferable representations that excel in few-shot and zero-shot BCSD scenarios. Despite these advances, function inlining remains a particularly stubborn obstacle: when the compiler replaces a function call with the body of its callee, the resulting binary can differ dramatically from any single "un-inlined" version.

To mitigate the effects of inlining on BCSD, several CFG expansion strategies have been proposed. BinGo and BinGo-E (Chandramohan et al., 2016; Xue et al., 2019) perform selective inlining-simulation by recursively expanding callee CFGs according to manually tuned heuristics (e.g., function size thresholds, coupling scores). Asm2Vec's authors adapted a similar approach, inlining only one layer of callees and pruning functions that exceed a length limit. More recently, OpTrans (Sha et al., 2025) proposed function-level heuristics, such as size, call-frequency thresholds, and stack size to identify inlining candidates. These manually defined heuristics help strike a balance between search accuracy and scalability, but they are inherently brittle: they miss many inlined functions and sometimes inline callees that should remain separate, resulting in low recall or precision.

Other works have tackled inlining from different angles. BINO (Binosi et al., 2023) introduces a fingerprinting framework to recognize inlined methods of C++ template classes by capturing both syntactic/semantic features and CFG structure, then matching via subgraph isomorphism. This approach achieves good precision and recall on known template methods but does not address arbitrary functions and incurs significant computational cost as the fingerprint database grows. ReIFunc (Lin et al., 2024) also leverages subgraph isomorphism and deep learning to identify recurring inline functions (RIFs) across binaries by detecting repeated basic-block patterns and then using a neural model to determine function origins. Although ReIFunc can locate inlined regions with high precision, it relies on expensive graph matching.

Meanwhile, O2NMatcher (Jia et al., 2022) and CI-Detector (Jia et al., 2024) focus on "1-to-n" matching for binary-to-source and binary-to-binary function mapping, respectively. O2NMatcher trains a multi-label classifier to predict which call sites will be inlined under various compilation settings, then generates source-function sets (SFSs) that represent all func-

tions merged into an inlined binary function. CI-Detector organizes binary functions into cross-inlining patterns and uses GNNs over attributed CFGs to compute similarity across inlining transformations. While both advance the state-of-the-art in identifying when and how functions are inlined, O2NMatcher's dependence on source-level information renders it inapplicable in purely binary-to-binary scenarios, and CI-Detector is an end-to-end cross-inlining embedding pipeline that does not explicitly discover inlined functions.

Taken together, prior works focusing on function inlining address distinct research problems:

- BINO and RelFunc intend to detect and recover the bodies (and, if possible, names) of functions that have been inlined into callers.
- O2NMatcher handles 1 to n binary to source mappings caused by inlining, by expanding the source side into "multi-function sets," then applying a standard 1 to 1 matcher.
- CI-Detector directly computes a similarity score between two binary functions when either (or both) may contain different inlining patterns, without explicitly recovering inlined bodies first.
- BinGo, Asm2Vec, and OpTrans expand certain callsites to normalize assembly functions in terms of function inlining.

Similar to BinGo, Asm2Vec, and OpTrans, FIN addresses function inlining by expanding CFGs rather than by post-hoc detection. To our knowledge, FIN is a pioneer work in predicting compiler inlining decisions explicitly for function inlining normalization. We chose CFG expansion over "detect-and-remove" schemes (e.g., BINO or ReIFunc) for two reasons: first, once inlining occurs, subsequent optimizations, such as constant folding, dead-code elimination, and common subexpression elimination, merge caller and callee code so tightly that there is no clean subgraph to delete; attempting removal risks losing or corrupting instructions. Also, false positive boundaries would delete code that remains semantically essential, whereas expansion simply duplicates the callee's graph without ever deleting original code. Second, detect-and-remove methods rely on expensive subgraph matching across large CFGs, which does not scale to thousands of functions. In contrast, our Random Forest classifier enables fast, large-scale normalization without sacrificing completeness.

7. Conclusion

Our study delves into the complexities of function inlining and introduces a novel solution, FIN, that enhances BCSD by intelligently expanding function CFGs. We identified the substantial impact of function inlining on BCSD performance and handcrafted a set of features to predict appropriate callees for CFG expansion. By expanding CFGs based on these predictions, we achieved significant improvements in binary code representation learning techniques. Our research also highlights the importance of the average distance of a callee function from its callers as a critical factor for CFG expansion. Additionally, we developed a tool to generate ground truth data, facilitating further research on the challenges of function inlining in cross-optimization BCSD. Our experiments showed that while CFG expansion is effective, it may not be sufficient to overcome all cross-optimization BCSD challenges. Therefore, in our future work, we aim to develop a representation learning approach that can more effectively incorporate the information added by CFG expansion.

8. Acknowledgment

This research is supported by Defence Research and Development Canada (contract no. W7701-217332), NSERC Discovery Grants (RGPIN-2024-04087), NSERC DND Supplement (DGDND-2024-04087), and Canada Research Chairs Program (CRC-2019-00041).

References

Ahn, S., Ahn, S., Koo, H., Paek, Y., 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning, Association for Computing Machinery, New York, NY, USA. p. 361–374. URL: https://doi.org/10.1145/3564625.3567975, doi:10.1145/3564625.3567975.

Bendersky, E., 2025. pyelftools: A Python library for parsing ELF files and DWARF debugging information. https://github.com/eliben/pyelftools. Accessed: March 05, 2025.

Binosi, L., Polino, M., Carminati, M., Zanero, S., 2023. BINO: Automatic recognition of inline binary functions from template classes. Computers & Security 132, 103312. URL: https://www.sciencedirect.

com/science/article/pii/S0167404823002225, doi:https://doi.org/10.1016/j.cose.2023.103312.

Breiman, L., 2001. Random forests. Machine learning 45, 5–32. doi:10.1023/A:1010933404324.

Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C.Y., Tan, H.B.K., 2016. BinGo: cross-architecture cross-OS binary search, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 678–689. URL: https://doi.org/10.1145/2950290.2950350, doi:10.1145/2950290.2950350.

Chen, W., Chung, Y.C., 2022. Profile-Guided optimization for Function Reordering: A Reinforcement Learning Approach, in: 2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 2326–2333. doi:10.1109/SMC53654.2022.9945280.

Ding, S.H.H., Fung, B.C.M., Charland, P., 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), pp. 472–489. doi:10.1109/SP.2019.00003.

Gu, Y., Shu, H., Kang, F., 2023. BinAIV: Semantic-enhanced vulnerability detection for Linux x86 binaries. Computers & Security 135, 103508. URL: https://www.sciencedirect.com/science/article/pii/S0167404823004182, doi:https://doi.org/10.1016/j.cose.2023.103508.

Guo, J., Zhao, B., Liu, H., Leng, D., An, Y., Shu, G., 2023. DeepDual-SD: deep dual attribute-aware embedding for binary code similarity detection. International Journal of Computational Intelligence Systems 16, 35.

Haq, I.U., Caballero, J., 2021. A Survey of Binary Code Similarity. ACM Comput. Surv. 54. URL: https://doi.org/10.1145/3446371, doi:10.1145/3446371.

Hex-Rays, 2024a. Ida pro 8.0 release notes. URL: https://hex-rays.com/products/ida/news/8_0. accessed: 2024-05-29.

Hex-Rays, 2024b. Ida pro sdk documentation: Operand types. URL: https://hex-rays.com/products/ida/support/sdkdoc/group__o__.html. accessed: 2024-05-29.

Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., Gu, D., 2018. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 104–114. doi:10.1109/ICSME.2018.00019.

Jia, A., Fan, M., Jin, W., Xu, X., Zhou, Z., Tang, Q., Nie, S., Wu, S., Liu, T., 2023. 1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis 32. URL: https://doi.org/10.1145/3561385, doi:10.1145/3561385.

Jia, A., Fan, M., Xu, X., Jin, W., Wang, H., Liu, T., 2024. Cross-Inlining Binary Function Similarity Detection, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3597503.3639080, doi:10.1145/3597503.3639080.

Jia, A., Fan, M., Xu, X., Jin, W., Wang, H., Tang, Q., Nie, S., Wu, S., Liu, T., 2022. Comparing One with Many–Solving Binary2source Function Matching Under Function Inlining. arXiv preprint arXiv:2210.15159 .

Khoshgoftaar, T.M., Golawala, M., Hulse, J.V., 2007. An Empirical Study of Learning from Imbalanced Data Using Random Forest, in: 19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007), pp. 310–317. doi:10.1109/ICTAI.2007.46.

Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y., 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. IEEE Transactions on Software Engineering , 1–23doi:10.1109/TSE.2022.3187689.

Li, M.Q., Fung, B.C.M., Charland, P., Ding, S.H.H., 2021. A novel and dedicated machine learning model for malware classification, in: Proceedings of the 16th International Conference on Software Technologies, SCITEPRESS - Science and Technology Publications. doi:10.5220/0010518506170628.

Li, Z., Liu, H., Shan, R., Sun, Y., Jiang, Y., Hu, N., 2023. Binary Code Similarity Detection: State and Future, in: 2023 IEEE 12th International Conference on Cloud Networking (CloudNet), pp. 408–412. doi:10.1109/CloudNet59005.2023.10490019.

Lin, W., Guo, Q., Yu, D., Yin, J., Gong, Q., Gong, X., 2024. ReIFunc: Identifying Recurring Inline Functions in Binary Code, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 670–680. doi:10.1109/SANER60148.2024.00074.

Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S., 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. IEEE Transactions on Software Engineering 43, 1157–1177. doi:10.1109/TSE.2017.2655046.

Luo, Z., Wang, P., Wang, B., Tang, Y., Xie, W., Zhou, X., Liu, D., Lu, K., 2023. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search, in: Proceedings 2023 Network and Distributed System Security Symposium, Internet Society, Reston, VA. doi:10.14722/ndss.2023.24415.

Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D., 2022. How Machine Learning Is Solving the Binary Function Similarity Problem, in: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA. pp. 2099–2116. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli.

Molloy, C., Charland, P., Ding, S.H.H., Fung, B.C.M., 2022. JARV1S: Phenotype Clone Search for Rapid Zero-Day Malware Triage and Functional Decomposition for Cyber Threat Intelligence, in: 2022 14th International Conference on Cyber Conflict: Keep Moving! (CyCon), pp. 385–403. doi:10.23919/CyCon55549.2022.9811078.

Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B., 2020. Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 .

Project, G., 2024. Gnu operating system. URL: https://www.gnu.org/. accessed: 2024-05-29.

Sha, Z., Lan, Y., Zhang, C., Wang, H., Gao, Z., Zhang, B., Shu, H., 2025. Optrans: enhancing binary code similarity detection with function inlining re-optimization. Empirical Software Engineering 30, 49.

Shalev, N., Partush, N., 2018. Binary Similarity Detection Using Machine Learning, in: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, Association for Computing Machinery, New York, NY, USA. p. 42–47. URL: https://doi.org/10.1145/3264820.3264821, doi:10.1145/3264820.3264821.

Sun, H., Shu, H., Kang, F., Guang, Y., 2023. ModDiff: Modularity Similarity-Based Malware Homologation Detection. Electronics 12. URL: https://www.mdpi.com/2079-9292/12/10/2258, doi:10.3390/electronics12102258.

Sun, R., Guo, S., Guo, J., Li, W., Zhang, X., Guo, X., Pan, Z., 2024. GraphMoCo: A graph momentum contrast model for large-scale binary function representation learning. Neurocomputing 575, 127273. URL: https://www.sciencedirect.com/science/article/pii/S0925231224000444, doi:https://doi.org/10.1016/j.neucom.2024.127273.

Theodoridis, T., Grosser, T., Su, Z., 2022. Understanding and exploiting optimal function inlining, in: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, NY, USA. p. 977–989. URL: https://doi.org/10.1145/3503222.3507744, doi:10.1145/3503222.3507744.

Wang, H., Gao, Z., Zhang, C., Sha, Z., Sun, M., Zhou, Y., Zhu, W., Sun, W., Qiu, H., Xiao, X., 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 503–515. URL: https://doi.org/10.1145/3650212.3652145, doi:10.1145/3650212.3652145.

Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C., 2022. JTrans: Jump-Aware Transformer for Binary Code

Similarity Detection, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 1–13. URL: https://doi.org/10.1145/3533767.3534367, doi:10.1145/3533767.3534367.

Wang, S., Wu, D., 2017. In-memory fuzzing for binary code similarity analysis, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 319–330. doi:10.1109/ASE.2017.8115645.

Xu, X., Feng, S., Ye, Y., Shen, G., Su, Z., Cheng, S., Tao, G., Shi, Q., Zhang, Z., Zhang, X., 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 1106–1118. URL: https://doi.org/10.1145/3597926.3598121, doi:10.1145/3597926.3598121.

Xue, Y., Xu, Z., Chandramohan, M., Liu, Y., 2019. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. IEEE Transactions on Software Engineering 45, 1125–1149. doi:10.1109/TSE.2018.2827379.

Yu, L., Lu, Y., Shen, Y., Huang, H., Zhu, K., 2021. Bedetector: A two-channel encoding method to detect vulnerabilities based on binary similarity. IEEE Access 9, 51631–51645. doi:10.1109/ACCESS.2021.3064687.

Zhao, D., Lin, H., Ran, L., Han, M., Tian, J., Lu, L., Xiong, S., Xiang, J., 2019. CVSkSA: cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. Softw. Qual. J. 27, 1045–1068. doi:10.1007/s11219-018-9435-5.

Zhao, P., Amaral, J.N., 2004. To Inline or Not to Inline? Enhanced Inlining Decisions, in: Rauchwerger, L. (Ed.), Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 405–419.