INVESTIGATING THE EFFECT OF PEEPHOLE OPTIMIZATIONS ON BINARY CODE DIFFERENCES

Xiaolei Ren

INVESTIGATING THE EFFECT OF PEEPHOLE OPTIMIZATIONS ON BINARY CODE DIFFERENCES

by

XIAOLEI REN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2023

To my dear mother and father who support me to explore this wonderful world.

ACKNOWLEDGEMENTS

As I reflect on my academic journey, I am filled with gratitude towards my parents for their unwavering support, encouragement, and love. Without their guidance and belief in me, I would not have achieved what I have today. Their trust in my abilities has been a constant source of inspiration and motivation.

I would also like to extend my heartfelt appreciation to my supervisors, Dr. Yu Lei and Dr. Jiang Ming, for their invaluable guidance, patience, encouragement, and inspiration. Thanks to them, I have become an independent thinker and decision-maker.

I am also deeply grateful to my committee members, Dr. Hao Che and Dr. Junzhou Huang, for generously sharing their time and valuable insights. Their feedback has been instrumental in shaping my research and academic pursuits.

Lastly, I would like to express my gratitude to my friends, teammates, and collaborators. Your unwavering support and encouragement have given me the opportunities to challenge myself and strive for greater achievements.

I am truly thankful to everyone who has helped me along this journey. Time has flown by so quickly, but the impact of your support and guidance will stay with me for a lifetime.

August 15, 2023

ABSTRACT

INVESTIGATING THE EFFECT OF PEEPHOLE OPTIMIZATIONS ON BINARY CODE DIFFERENCES

XIAOLEI REN, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professors: Dr. Yu Lei and Dr. Jiang Ming

Binary diffing is a technique used to compare and identify differences or similarities in executable files without access to source code. The potential applications of binary diffing in various software security tasks, such as vulnerability search, code clone detection, and malware analysis, have generated a vast body of literature in recent years. One of the recurring themes in binary diffing research is the evaluation of its resilience against the impact of compiler optimization, which is the most common source of syntactic differences in binary code. Despite that most binary diffing tools claim that they are immune to compiler optimization, recent studies have highlighted the need for the research community to revisit this claim, particularly regarding non-default optimization settings and function inlining.

In this study, we investigate the effect of peephole optimization on binary diffing analysis. Peephole optimization is a feature of mainstream compilers that allows local rewriting of the input program. It replaces instruction sequences within a window (i.e., peephole) with shorter, faster, or functionally equivalent instruction sequences. Our research reveals that peephole optimization primarily affects binary code differences at the intra-procedural level, which contradicts the assumptions made by basic-block-centric comparison approaches. We conducted systematic experiments using LLVM's unit test suite. We also customized Alive2, an LLVM translation validation tool, to isolate the impact of peephole optimization from the overall optimization process.

Our investigation determines the pervasiveness of peephole optimization in the resulting compiled code and explores its effects on current binary diffing techniques. The noticeable decline in performance highlights the importance of considering peephole optimization in the analysis and improvement of binary diffing methodologies. Therefore, our findings suggest that researchers and practitioners should consider the impact of peephole optimization when developing and evaluating binary diffing tools. Further research is necessary to address this challenge and improve the effectiveness of binary diffing in various software security tasks.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

In the contemporary digital ecosystem, marked by a high degree of interconnectivity, software security has emerged as a critical concern for various stakeholders, including individuals, corporations, and governments. This heightened importance can be attributed to the rapid proliferation of Internet of Things (IoT) devices and the escalating threats posed by malicious software, or malware [5,6].

A considerable portion of system programs, IoT device firmware, and malware are developed using C/C++ programming languages, resulting in their distribution in binary code form. Binary code, a low-level representation of software, often lacks the high-level language information required for comprehensive analysis. Consequently, security analysts are faced with the challenge of assessing software security without access to the source code in numerous real-world scenarios [7,8,9].

In such situations, binary diffing serves as a valuable tool for comparing and differentiating between two or more binary code pieces. This technique facilitates the identification of similarities and differences among multiple instances of binary code, enabling a wide array of security tasks. These tasks include vulnerability search [10, 11, 12, 13, 14, 15, 16], security patch analysis [17, 18, 19], code clone detection [20,21,22,23,24], and malware analysis [25,26,27,28,29,30,31]. By effectively identifying potential security threats and promoting the development of suitable countermeasures, binary diffing techniques substantially contribute to enhancing overall software security.

1.1 Binary Diffing

Binary diffing is a technique used to compare differences or similarities between two binary code versions, uncovering valuable information even without the source code. For instance, analyzing the similarities between different binary codes can reveal underlying relationships, such as code clones and similar malware lineage. Therefore, the benefits of binary diffing have led to wide adoption by various software security analysis tasks.

Binary diffing involves identifying functions, addressing indirect calls and virtual function tables, as well as modifying calling conventions. To achieve this, binary comparison tools parse binary code and gather relevant information, such as functions, control flow graphs (CFGs), function flow graphs, basic blocks, edges, and instructions. After collecting the data, these binary comparison tools employ CFG-based graph comparison and function comparison algorithms, delving into the basic blocks. This process results in a comprehensive difference analysis between the two binary files, taking into consideration all the aforementioned code representations. To further enhance the comparison analysis results, some binary comparison approaches integrate with advanced analysis toolkits such as IDA Pro [32], enabling access to higher levels of abstraction and filtering out low-level noise.

1.2 The Importance of Binary Diffing in Software Security

The significance of analyzing and understanding binary code behavior has surged in recent years, mainly due to the rapid growth of system programs, Internet of Things (IoT) device firmware, and various software, which heighten software security threats [5,6]. Binary diffing can analyze the similarities between different binary codes, revealing crucial underlying relationships. For example, comparing a binary file with its patched version can expose a fixed vulnerability that attackers might exploit to compromise the unpatched programs. Similarly, detecting similarities between an intellectual property-protected program and a suspicious program could indicate software plagiarism. In malware analysis, binary diffing helps classify malware samples with shared functions, allowing security professionals to concentrate on new variants and enhance overall software security.

1.2.1 Vulnerability Search and Security Patch Analysis

Vulnerability search [10, 11, 12, 13, 14, 15, 16] is the process of identifying weaknesses and flaws in a software system that can be exploited by malicious actors. The purpose of vulnerability search is to expose and address these issues proactively to secure the system before any potential breaches or attacks occur. Binary diffing is an effective technique used in vulnerability search that compares two binary files to detect similarities or differences. In the context of software security, binary diffing is often applied to identify vulnerabilities in software patches or different versions of the same application. This method enables security analysts to uncover newly introduced or fixed vulnerabilities by comparing a patched binary with its previous unpatched version. It can also be used to find similarities between a known vulnerable binary and other binaries, revealing potential shared vulnerabilities.

Security patch analysis [17, 18, 19] is a vital aspect of vulnerability search, focusing on the evaluation of patches applied to software to address identified security vulnerabilities. By examining the differences between the two versions, researchers can understand the specific fixes applied to address security issues, assess the effectiveness of the patch, evaluate its potential impact on system performance, and even reverse-engineer the vulnerability. Gaining insights into potential exploits helps security experts better understand the threats they face and develop more robust defenses against them. In this way, security patch analysis complements the vulnerability search process, providing a comprehensive understanding of software vulnerabilities and their remediation strategies.

1.2.2 Code Clone Detection

Code clone detection is the process of identifying identical or similar code fragments in different binary programs, which can arise from code reuse, duplication, or plagiarism. Detecting code clones is essential in various scenarios, such as understanding software evolution, refactoring, and detecting potential security vulnerabilities. If the cloned code contains any exploitable flaws, it may pose a significant risk to the system's security, as the vulnerability may be present in multiple instances [20, 21, 22, 23, 24]. Binary diffing is a valuable technique for code clone detection, as it allows comparing binary files directly without requiring access to the source code. This is particularly useful when working with proprietary or obfuscated binaries, where source code may not be available. By comparing the binary files of a new application and a vulnerable component, researchers can identify similarities and differences that not only help detect code clones but also enable security experts to determine whether the same vulnerability exists in the new software system.

1.2.3 Malware Analysis

Malware analysis [25, 26, 27, 28, 29, 30, 31] is a crucial process that involves examining and dissecting malicious software to understand its behavior, functionality, and potential impact on targeted systems. This analysis is essential for developing effective countermeasures and mitigating risks posed by malware. A primary objective of malware analysis is identifying and classifying malware samples based on shared functionalities, which assists security professionals in detecting new variants of known malware families and focusing their efforts on understanding and combating emerging threats. Binary diffing plays a critical role in malware analysis, as it enables researchers to compare different binary files and identify similarities and differences between them. By analyzing the shared functionalities between malware samples, binary diffing aids in classifying and categorizing malware into families and detecting new variants or evolutions of existing malware. This capability is particularly valuable when dealing with obfuscated or encrypted malware, where traditional analysis techniques may struggle to extract key information about the malware.

1.3 Technical Challenges of Binary Diffing

The field of binary diffing has experienced significant growth and research interest in recent years. As evidenced by a survey conducted by Haq et al. [33], over 100 published papers in top computer science venues have covered various aspects of binary diffing. These contributions span multiple disciplines, including computer security, software engineering, programming languages, and machine learning. These papers address problems including vulnerability search, malware analysis, patch inspection, plagiarism detection, and de-anonymizing code authors, using different code representations and semantic similarity measures. Despite its versatile applications, the accuracy of binary diffing is subject to several challenges from code representation, the similarity measurement of code presentation, and compiler optimization. Among them, compiler optimization makes it difficult to define a semantics-aware code representation that effectively captures the similarities between programs [9].

1.3.1 How to Define a Code Representation

In the realm of software security, binary diffing plays a vital role in facilitating the in-depth analysis and comparison of similarities between different programs. To effectively carry out binary diffing, it is essential to carefully select and define appropriate code representations that enable the identification of connections and similarities between similar programs. To achieve this goal, researchers have explored various code representation techniques, including control flow graphs (CFGs), functions, basic blocks, and instruction n-grams [33]. These representations serve as a foundation for identifying similarities between binary files, allowing researchers to pinpoint code sections with potential modifications or vulnerabilities.

1.3.1.1 Defining Code Representation

A control flow graph (CFG) is a directed graph that represents all potential execution paths within a program. In binary diffing analysis, CFG can assist researchers in examining the structural differences between two binary files. However, the representation of CFG can be influenced by factors such as the compiler, optimization options, and the program's architecture, thereby posing challenges for binary diffing analysis. For instance, function inlining optimization integrates callee functions into the caller's body, subsequently impacting the CFG.

Functions serve as logical organizational units of code, typically designed to perform specific tasks. In binary diffing analysis, representing code as functions can aid researchers in identifying potential modifications and vulnerabilities. Nevertheless, in certain cases, similarities between functions may not necessarily indicate functional similarity, requiring further analysis and verification.

A basic block represents a linear organization of instruction sequences, wherein each instruction is executed in order, without jumps or branches. Basic blocks can help researchers analyze similarities between instruction sequences. However, factors like compiler optimizations can affect the similarity between basic blocks, making binary diffing analysis more challenging. For example, peephole optimization can replace a basic block's instruction sequence with simplified instructions, thus altering the basic block and its instructions.

Instruction n-grams [34] provide a method for representing local features of instruction sequences. In binary diffing analysis, instruction n-grams can help researchers pinpoint potential similarities and differences. However, factors such as the compiler and optimization options can impact the analysis using instruction n-grams, resulting in false positives or false negatives.

1.3.1.2 Challenges in Defining Code Representation

This is primarily due to the low-level and intricate nature of binary code, which lacks the high-level language information found in source code, such as variable names, data structures, and types. The absence of this information makes it significantly more difficult to discern the underlying logic of a program, which in turn hampers the accuracy of binary diffing techniques.

One of the primary challenges in binary diffing is the absence of symbolic information in the code representation. In source code, meaningful names are assigned to variables, functions, and data structures, providing context and facilitating comprehension. However, binary code does not retain this information, making it more challenging to identify and analyze corresponding program components. As a result, researchers must rely on alternative methods, such as pattern matching and statistical analysis, to bridge this gap.

Another challenge in binary diffing is the presence of compiler optimizations, which can alter the structure and content of the generated binary code, consequently affecting the code representation. For instance, function inlining optimization impacts the function scope and control flow, while peephole optimization affects basic blocks. These optimizations can vary depending on the compiler used, the optimization level selected, and the target architecture, resulting in different binary representations for the same source code. This variability can hinder the process of accurately comparing and identifying similarities between binary files.

Additionally, the handling of different instruction sets and architectures presents further challenges in defining effective code representations. Different processor architectures have unique instruction sets and conventions, which can lead to dissimilar binary code representations for equivalent functionality. Malware originating from the same virus family is distributed across various computing systems, including PC software and IoT software, spanning architectures such as x86-32, x86-64, ARM, and MIPS. Consequently, researchers must account for these differences and develop techniques to normalize and compare code across various architectures.

1.3.2 How to Measure Similarities Between Code Representations

The evaluation of code representation similarities is a multifaceted task that requires the use of various evaluation metrics, including graph comparison, black-box testing, and equivalence checking through software verification.

Graph comparison techniques are employed to assess structural similarities between code representations, such as control flow graphs or data flow graphs. These methods involve comparing nodes and edges, which represent program components and their relationships, respectively. However, graph comparison can be computationally expensive and may not always provide an accurate representation of functional similarity, as different graph structures can potentially lead to the same functionality.

Black-box testing focuses on evaluating the functionality of programs based on their responses to specific inputs, without examining their internal structures or implementation details. This approach can be advantageous in assessing code similarity, as it solely relies on the observable behavior of the programs. However, black-box testing may not be exhaustive, as it is often impractical to test every possible input combination. Additionally, it may not detect similarities in code that are not executed during testing.

Equivalence checking utilizes software verification techniques to determine if two code representations are functionally identical. This method typically involves constructing mathematical models of the programs and proving their equivalence using formal methods, such as theorem proving or model checking. While equivalence checking can provide rigorous guarantees of code similarity, it can be time-consuming and may not scale well to large or complex programs.

It is essential to consider the inherent challenges and limitations of these evaluation metrics when assessing similarities between code representations in binary diffing analysis. Researchers must balance the trade-offs between accuracy, scalability, and computational complexity while selecting the most suitable evaluation methods for their specific use cases. Furthermore, leveraging a combination of these evaluation metrics can help achieve a more comprehensive understanding of the similarities between code representations, ultimately enhancing the effectiveness of binary diffing analysis.

1.3.3 Compiler Optimizations Affecting Binary Diffing

The study of compiler optimization has garnered substantial attention from researchers in the fields of software engineering and programming languages. This growing interest is primarily attributed to the potential implications of binary differences on aspects such as software security and performance.

Compiler optimization refers to the process of refining the generated code by a compiler to improve the software's overall performance, reduce its size, or minimize its resource consumption. However, these optimizations may result in changes to the binary code, which can lead to binary code differences. Recent research has highlighted the importance of understanding the various factors that contribute to binary differences during compilation, such as the type and version of the compiler, the target architecture of the executable file, and compiler optimization techniques [33, 35].

In their study, Kim et al. [35] systematically analyzed the impact of these factors on binary differences. They discovered that compiler optimization is a common and critical determinant of these differences. This finding implies that the choice of optimization techniques and levels can significantly influence the resulting binary code. Consequently, it is crucial for security experts to understand the implications of compiler optimization techniques on binary differences to make informed decisions during the binary diffing process.

On the other hand, the prevailing belief in the field of binary diffing is that most tools are capable of producing satisfactory results across a range of compiler optimization levels. As a result, the extensive research conducted in this area should enable a comprehensive understanding of how different code representations and comparison methodologies can adequately discern the syntactic differences that arise from compiler optimization. However, recent studies by Ren et al. [36] and Jia et al. [37] have called these assumptions into question, positing that the impact of compiler optimization on binary code differences may have been significantly underestimated.

Ren et al.'s BinTuner [36] investigates the effects of non-default optimization settings by adopting an iterative compilation technique [38,39]. This method searches for near-optimal optimization sequences that maximize binary code differences. When compared to leading binary diffing tools, there is a noticeable decline in accuracy when these tools are applied to binary code generated by BinTuner. This finding underscores the need for a more in-depth examination of the impact of compiler optimization on binary diffing tools.

Moreover, Jia et al.'s study [37] highlights that many binary function comparison approaches fail to account for the consequences of function inlining optimization. This optimization technique integrates callee functions into the body of the caller, and can account for as much as 70% of inlined functions when using O3. Existing binary function matching strategies, however, can result in a reduction of up to 30% in code search performance and a decrease of 40% in vulnerability detection effectiveness. This evidence further emphasizes the necessity for ongoing research in this area to better comprehend the subtleties of binary diffing tools and the ramifications of compiler optimization on their efficacy.

Previous researchers have explored how compiler optimizations can influence binary diffing, but there is still much to uncover about the complex relationships between these optimizations and the accuracy of binary comparison techniques. By examining specific optimization methods, such as peephole optimization, we can gain a deeper understanding of the challenges faced by binary diffing tools in the presence of optimized code.

1.4 The Importance of Studying the Impact of Peephole Optimization on Binary Diffing

The investigation of the ramifications of peephole optimization on binary diffing possesses substantial importance within the realms of computer science and system security. Peephole optimization, a compiler optimization technique, primarily concentrates on localized enhancements within a confined scope of instructions, frequently culminating in the generation of more efficient code. A critical analysis of the consequences stemming from peephole optimization on binary diffing is essential for several reasons:

Augmenting Security Analysis: A thorough comprehension of the impacts of peephole optimization can facilitate the advancement of malware detection and analysis methodologies. By pinpointing the manner in which peephole optimization modifies binary code, security specialists can conceive more efficacious approaches for monitoring malware progression and discovering commonalities between distinct malicious binaries, even in cases where obfuscation or packing is employed.

Fostering Compiler Technique Progression: The examination of the effects of peephole optimization on binary diffing may contribute to the evolution of increasingly sophisticated compiler optimization methodologies. As researchers acquire a more profound understanding of the complex interconnections between code optimizations and binary representations, they can devise novel strategies for optimizing compilers to a greater extent, ultimately resulting in the creation of more efficient and secure software.

Enabling Reverse Engineering and Debugging Proficiency: Acquiring insight into the implications of peephole optimization on binary diffing can support reverse engineers and debuggers in accomplishing their objectives. By discerning the patterns and transformations introduced through peephole optimization, these professionals can more precisely decode binary code, pinpoint discrepancies, and detect issues within software systems.

In summary, the significance of exploring the impact of peephole optimization on binary diffing resides in its potential to amplify system security, promote compiler technique advancements, and streamline reverse engineering and debugging endeavors. By immersing themselves in this research domain, computer science and system security experts can contribute to the cultivation of increasingly efficient, secure, and dependable software systems.

1.4.1 How Peephole Optimizations Affecting Binary Diffing

As compiler technology advances, modern compilers, like LLVM, offer an extensive array of peephole optimization options, providing over a thousand such alternatives through the *InstCombine* pass [3, 4]. This optimization technique enables more efficient and streamlined code generation. However, this process can inadvertently complicate binary diffing, the comparative analysis of binary executables to identify similarities and differences. The transformations introduced by peephole optimizations may result in distinct binary representations of functionally equivalent code, consequently increasing the difficulty of accurately discerning and correlating the underlying algorithms or data structures. Therefore, understanding the impact of peephole optimizations on binary diffing is vital for the development of robust and precise reverse engineering techniques and tools in the realm of software security. The impact of peephole optimizations on binary code is also an intriguing topic, and we will explore later their influence on binary code and its subsequent analysis through two illustrative examples in the motivation section.

1.5 Approach

Unlike previous studies that focused on non-default optimization options [36] or inter-procedural effects [37], in this paper, we zoom in on a local code optimization technique: peephole optimization [40]. It recurs from O1 to O3 settings and enables local modifications to the input program. Specifically, it can replace instruction sequences within a specific window (i.e., peephole) with more concise or faster equivalent instructions [41,42,43,44]. LLVM compiler optimizations are achieved through a sequence of *Passes* [1,2], which are responsible for executing code transformations and optimizations. Peephole optimization is particularly effective in generating code customized based on specific hardware architectures. It can leverage architecture-specific instructions and features, thereby improving code execution efficiency and better utilizing available hardware resources.

It is important to note that from the user's perspective, LLVM does not provide options to separately disable or enable peephole optimizations, such as (-fno-peephole). For example, it is not possible to specifically enable peephole optimizations in an unoptimized setting (O0), nor activate all default optimizations (O3) while only disabling peephole optimizations and keeping other optimizations unchanged. In other words, users cannot compile different binary codes containing only peephole optimizations or no peephole optimizations by setting the disabling or enabling of peephole optimizations at compile time. Therefore, analyzing the impact of peephole optimizations on binary code remains an unresolved technical challenge.

1.5.1 Implementation

Examining the impact of peephole optimization on a binary code segment within the final optimized binary code poses a considerable challenge, owing to the amalgamation of optimization outcomes at the binary level. To comprehensively evaluate the effect of peephole optimization on binary code discrepancies, we propose a two-step method that utilizes the LLVM unit test suite and Alive2 [45], an LLVM translation verification tool. This approach ensures an analysis of the optimization process and its implications on binary code differences.

In the first step, we concentrate on isolating the specific impact of peephole optimization within the broader optimization process. To achieve this, we focus on how peephole optimization *InstCombine Pass* [3, 4] changes the intermediate representation (IR), conducting a series of methodical experiments on IRs changes. By identifying and examining the transformations applied solely by the *InstCombine Pass*, we can separate the influence of peephole optimization from other optimization techniques.

In the second step, we delve into the evaluation of the influence of peephole optimization on basic blocks. By analyzing the changes in basic blocks resulting from the application of peephole optimization, we can assess its impact on binary diffing tools' performance. This granular analysis allows us to better understand the specific benefits and drawbacks of peephole optimization in relation to other optimization techniques.

Through this approach, we aim to provide an assessment of the impact of peephole optimization on binary code variance and offer valuable insights into its role within the broader optimization process. This information can be used to guide future research and development efforts in the field of compiler optimizations and system security.

1.6 Contributions

The utilization of peephole optimization introduces complications in the pair-wise comparison of basic blocks. This poses a challenge to numerous binary diffing tools that rely heavily on such comparisons as their fundamental framework. Despite this challenge, there is a lack of systematic study into the extent and impact of peephole optimization on existing binary diffing techniques. Our paper aims to fill this gap by investigating the following three research questions (RQs):

RQ1: How does peephole optimization affect the syntactic differences of binary code? Gaining a deep understanding of these effects is vital to the design of a robust binary diffing tool.

RQ2: To what extent does peephole optimization happen throughout the entire optimization process? This demonstrates whether peephole optimization prevails in the compiled code.

RQ3: What impact does peephole optimization have on existing binary diffing techniques? The performance degradation indicates the importance of considering peephole optimization.

The first challenge of our study is to identify the code changes that are *solely* caused by peephole optimization. To illustrate this point, we can consider LLVM's InstCombine pass, which performs peephole optimization in an iterative manner throughout the entire optimization process. Furthermore, this pass significantly interacts with other optimization passes, such as the simplifyCFG pass. Therefore, to answer RQ1 & RQ2, we leverage Alive2 [45], an LLVM translation validation tool, to track the effect of each InstCombine pass with a default optimization setting (e.g., O1∼O3) and LLVM's unit test suite as inputs. In particular, we take snapshots of pre- and post-optimization IR code that have equivalent semantics; then, we adopt Myers algorithm [46] to identify syntactic differences of basic block pairs. We present a new taxonomy to summarize such binary code changes at the intra-procedural level. By doing so, we can closely monitor the optimization pipeline and collect statistical data on the effect of peephole optimization.

In summary, by integrating the three research questions to conduct experiments and analyses, our main contributions are as follows:

We propose a novel classification method for summarizing intra-program-level binary code changes of this nature. By doing so, we can monitor the optimization process and analyze the ubiquity of peephole optimization effects (Chapter 5).

We present a feasible approach to observe the transformation process of peephole optimization on each function during the optimization process, addressing the challenge of isolating peephole optimization effects from the optimization process (Chapter 6).

Our findings indicate that peephole optimization is the most prevalent optimization pass, with an average invocation frequency ranging from 20.1% to 37.9%. Across all O1 to O3 levels, the average percentage of basic blocks modified by peephole optimization varies between 32.8% and 60.2%, with a maximum value of 76.3% (Chapter 8).

Our study demonstrates that peephole optimization affects current binary difference methods at both the basic block and control flow levels. The performance degradation underscores the importance of considering peephole optimization. BinDiff results indicate that peephole optimization can cause 50% of optimized instructions within basic blocks to be unmatchable and lead to modifications spanning multiple basic blocks. Jumps and Calls identify over 50% of the differences. These changes in basic blocks significantly impact DeepBinDiff's basic block embedding-based approach, resulting in a substantial number of unmatched basic blocks (Chapter 9).

CHAPTER 2

Background

In this section, we first provide a concise overview of binary difference analysis research. Next, we delve into the LLVM compiler optimization process and the origin and significance of peephole optimization from the compiler's perspective.

2.1 Binary Difference Analysis Research

The similarities between two pieces of different binary code can reveal underlying relationships, such as code clones and close malware lineage, even without access to the source code. As a result, binary diffing's multifaceted benefits have led to its widespread adoption in various software security analysis tasks. In recent years, the field of binary diffing has witnessed significant growth and research interest.

A survey by Haq et al. [33] highlights that over 100 published papers in top computer science venues have explored different aspects of binary diffing. These contributions encompass multiple disciplines, such as computer security, software engineering, programming languages, and machine learning. They address a range of problems, including vulnerability search, malware analysis, patch inspection, plagiarism detection, and de-anonymizing code authors, employing diverse code representations and semantic similarity measures.

The foundation of a binary diffing approach lies in defining a semantics-aware code representation that enables similar programs to exhibit close representations. Consequently, the representation and comparison of code semantics become crucial aspects of binary analysis.

At the syntactic level, most research papers focus on discernible binary code structures as code representations, including functions, basic blocks, loops, traces, control flow graphs (CFGs), and call graphs (CGs). The accuracy of detecting these representations depends on precisely locating their scope within the code.

In contrast, at the semantic level, methods for measuring code representation semantics are more diverse. They range from computationally expensive yet accurate approaches to scalable but less robust techniques. Some examples include: Symbolic execution, which represents input-output relationships as formulas and verifies their equivalence using a theorem prover [22, 25, 47, 48, 49]. Dynamic testing, which generates concrete inputs automatically to compare output values [13, 16, 50, 51, 52]. Basic block re-optimization, which normalizes syntactically different data-flow slices to facilitate scalable search [11, 53]. Descriptive statistic features, such as the number of transfer instructions, aim for fast matching of target functions among large-scale binaries (large-scale executable files) [14, 54].

Recent research has also explored the potential of deep learning and neural networks to learn the relationship between binary code snippets [20, 55, 56, 57]. For instance, Asm2Vec [20] learns the lexical semantic relationships within the x86/64 instruction set in a function scope. It identifies patterns such as Streaming SIMD Extensions (SSE) operands being related to SSE registers and the typical co-occurrence of file-related APIs.

Binary diffing hinges on the effective representation and comparison of code semantics, with various techniques employed at both the syntactic and semantic levels. By leveraging these techniques, researchers can better understand and identify similarities or differences between binary code snippets, ultimately improving the accuracy and efficiency of binary diffing approaches.

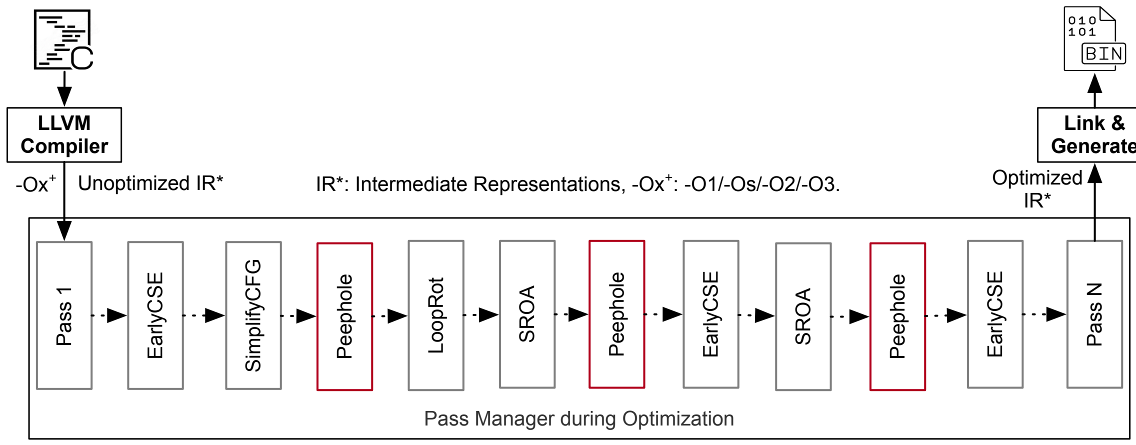


Figure 2.1: LLVM compiler optimizations utilize a sequence of *Passes* [1, 2] for code transformations and enhancements. *EarlyCSE*: identifies and removes repetitive subexpressions or operations within a function. *SimplifyCFG*: streamlines the program's Control Flow Graph (CFG) by eliminating redundant control structures, merging adjacent basic blocks, and removing unreachable code. *LoopRotate*: by rotating the loop body, it aligns the loop header with the most frequent exit path, reducing branches during execution, and improving code locality and cache utilization. *SROA* (Scalar Replacement of Aggregates): Enhances performance by breaking down aggregate data structures like arrays and structures into individual scalar variables. This promotes efficient memory access and further optimization opportunities for other LLVM passes, improving execution efficiency, especially in applications with intricate data structures and memory access patterns.

2.2 Optimization Processes in LLVM

Compiler optimization is an essential process in modern software development, aiming to enhance the performance of compiled programs and reduce resource consumption, such as memory usage and power consumption. This process involves a series of transformations applied to the intermediate representation (IR) of the source code to improve the quality of the final executable code while preserving the semantics of the original program. Compiler optimizations impact binary code and binary structures and make it difficult to define a semantics-aware code representation that effectively captures program similarities.

The LLVM compilation toolchain is a popular choice for compiler optimization due to its modular design, robustness, and support for a wide range of programming languages and target platforms. Fig. 2.1 shows LLVM provides a rich set of optimization passes organized and orchestrated by the Pass Manager, a crucial component in the LLVM toolchain. The Pass Manager's primary responsibility is to manage the scheduling and execution of optimization passes, ensuring that they are applied efficiently and effectively [1, 2]. It considers various factors, such as optimization levels, target-specific optimizations, and inter-pass dependencies, to determine the most appropriate order and combination of passes for a given input program. Each pass can be called multiple times, either to optimize the same function multiple times or to optimize different functions separately in each pass. For instance, Fig. 2.1 illustrates that the red-highlighted peephole optimization is called multiple times. Given that peephole optimization operates at the function level, multiple calls may indicate that the same function is optimized repeatedly or that multiple functions undergo separate optimizations. Therefore, when analyzing peephole optimization later, it is crucial to identify them based on functions and analyze multiple times peephole optimizations separately for different functions.

LLVM optimization passes can be broadly classified into two categories: local optimizations, which focus on improving individual basic blocks or functions, and global optimizations, which target the entire program or module. Examples of key optimization passes include loop unrolling, function inlining, and peephole optimization. Local optimizations aim to simplify and streamline code within a basic block or function by eliminating redundancies, applying algebraic simplifications, and replacing inefficient instruction sequences with more efficient alternatives. In contrast, global optimizations, such as interprocedural analysis and loop optimizations, address higher-level code structures and interactions between functions, seeking to optimize the overall program structure and control flow.

The Pass Manager [2] plays a crucial role in the compilation optimization process by overseeing the execution of various compilation passes responsible for analyzing and transforming the program's IR to generate optimized machine code for the target architecture. It manages to pass registration, scheduling, and dependencies, enabling a seamless and efficient compilation pipeline. The Pass Manager ensures that all essential analysis results are obtained before executing a pass, which is vital for the effectiveness of a transformation pass that often depends on the availability of accurate and up-to-date information about the program's data flow, control flow, and other properties. By guaranteeing that all necessary analyses are performed before a transformation passes, the Pass Manager contributes to the generation of highly optimized code.

Furthermore, the Pass Manager takes care of scheduling and handling dependencies between passes by establishing an order of execution that respects the interdependence of various passes, preventing potential conflicts or redundancies. This optimized scheduling allows the compilation pipeline to function efficiently, eliminating wasted resources and time.

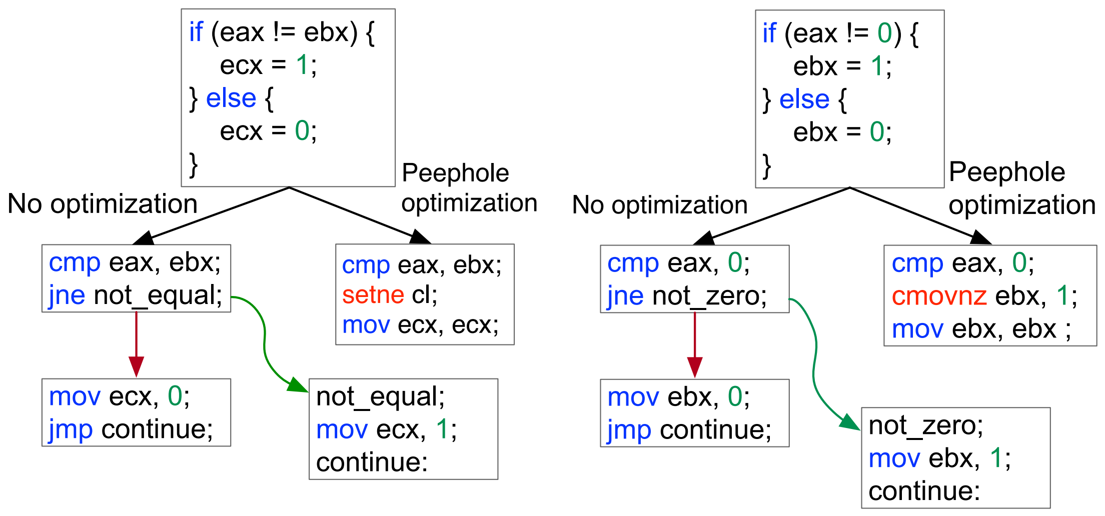
In the LLVM compilation toolchain, the Pass Manager serves as the scheduling driver, enabling the optimization of real-world programs through numerous optimization cycles when using the O3 level. This aggressive optimization level applies a wide range of techniques, such as loop unrolling, function inlining, and dead code elimination, to significantly improve the performance of the compiled code. For instance, the bzip2 program underwent 16,167 optimizations during the O3 level, resulting in a total IR file size of 22.08 GB.

The LLVM Pass's strategy is instrumental in evaluating the impact of individual peephole optimizations, given the compilation optimization process's inherent complexity involving numerous passes, analyses, and transformations. To effectively manage this complexity, the Pass Manager ensures proper execution, scheduling, and resource management, thereby optimizing machine code and enhancing performance and resource utilization in real-world applications through the LLVM compilation toolchain. In contrast to GCC, which generates a dump file containing final optimizations, this approach allows us to track changes to basic blocks and instructions within a function at each step of optimization, providing a more logical and detailed view of the process.

2.2.1 Peephole Optimization

Peephole optimization, first introduced by McKeeman in the early 1960s, is a fundamental technique in compiler optimization [40, 41, 42, 43]. It enhances code efficiency by analyzing and optimizing small instruction sequences, known as "peepholes". Typically consisting of 2 to 3 contiguous instructions, peephole optimization does not significantly increase the compiler's complexity or slow it down [42]. By examining a limited window of adjacent instructions, the compiler can eliminate redundant or inefficient code, thereby improving program performance. Over the years, peephole optimization has evolved, extending its scope from early assemblers to various code representations and providing over a thousand optimizations [58, 59].

Peephole optimization is a critical technique in the late stages of compilation, aiming to achieve optimal instruction selection. It employs thousands of rules to enhance code quality, eliminate redundancies, and normalize code. This technique excels at identifying and replacing common code patterns with more efficient alternatives like constant folding, strength reduction, and instruction substitution. Applying



```
if (eax != ebx) {
    ecx = 1;
} else {
    ecx = 0;
}
```

```
cmp eax, ebx;
jne not_equal;
```

```
mov ecx, 0;
jmp continue;
```

```
not_equal;
mov ecx, 1;
continue:
```

```
cmp eax, ebx;
setne cl;
mov ecx, ecx;
```

```
if (eax != 0) {
    ebx = 1;
} else {
    ebx = 0;
}
```

```
cmp eax, 0;
jne not_zero;
```

```
mov ebx, 0;
jmp continue;
```

```
not_zero;
mov ebx, 1;
continue:
```

```
cmp eax, 0;
cmovnz ebx, 1;
mov ebx, ebx ;
```

(a) Branch-free instruction (b) Conditional moves

Figure 2.2: Peephole Optimizations.

these optimizations at the peephole level brings notable improvements in program execution time, memory usage, and code quality. Moreover, peephole optimization is crucial for generating efficient code across different platforms by targeting specific instructions or patterns that exhibit suboptimal behavior. It tailors the optimization process to leverage the unique characteristics and capabilities of the underlying hardware. By significantly enhancing performance and reducing code size, peephole optimization plays a vital role in producing efficient and streamlined executable files. Its impact on the binary is reflected in code transformations at the intraprocedural level, including the impact of instruction sequences, basic blocks, and control flow edges.

Peephole optimization can lead to the complete modification of basic blocks, posing challenges to mainstream binary difference comparison methods, particularly those using basic blocks as code representations in their comparison approaches.

One example of peephole optimization is the use of branch-free instructions to eliminate conditional branches in certain scenarios, resulting in potentially more efficient code. In Fig. 2.2 (a), the `setne` instruction is an x86 assembly instruction that sets a byte-sized register to 1 if the previous comparison resulted in a “not equal” condition, and sets it to 0 otherwise. By using the `setne` instruction, it can avoid branching and directly set the value of a register based on the result of a comparison. This can lead to improved code execution and reduced branch mispredictions.

Another example is the use of conditional moves, as shown in Fig. 2.2 (b). The `cmovnz` instruction is an x86 assembly instruction that performs a conditional move based on the “not zero” condition. It copies the value from one register to another if the zero flag is not set, avoiding the need for explicit branching. Instead of branching based on a condition, the conditional move allows the value to be directly moved based on the condition, reducing pipeline stalls and improving performance. By employing such techniques, peephole optimization can significantly enhance the efficiency of the generated code.

CHAPTER 3

Motivation

In this chapter, we present the necessity of revisiting the impact of compiler optimization on binary code, show how significant peephole optimization transforms basic blocks with two examples, and discuss the current limitations of binary comparison tools in handling peephole optimizations.

3.1 The Imperative to Re-evaluate the Consequences of Compiler Optimization on Binary Code

In the field of binary comparison, a multitude of tools assert their adaptability to various compiler optimization configurations and have the capacity to deliver satisfactory results in the analysis of binary code across a wide range of optimization levels. As such, one anticipates that the extensive research conducted in this domain would effectively address crucial research questions regarding the selection of appropriate code representations and corresponding comparison methodologies that can efficiently discern the syntactic disparities emerging from compiler optimization.

Contrary to these expectations, recent investigations [36, 37] have raised concerns, insinuating that the impact of compiler optimization on binary code discrepancies has not been entirely addressed. Many binary analysis techniques struggle to cope with the changes introduced by compiler optimizations. Research has confirmed that function inlining leads to the merging of multiple functions, which hinders the comparison methods from accurately aligning function matches [37]. Similarly, peephole optimization replaces instruction sequences within basic blocks, resulting in comparison methods being unable to correspond to the fundamental matchable pairs, ultimately affecting the precision of the final comparison results. Consequently, further systematic compiler optimizations (such as peephole optimization) are deemed necessary.

3.2 Peephole Optimization Modifies Binary Code at Intra-procedural Level

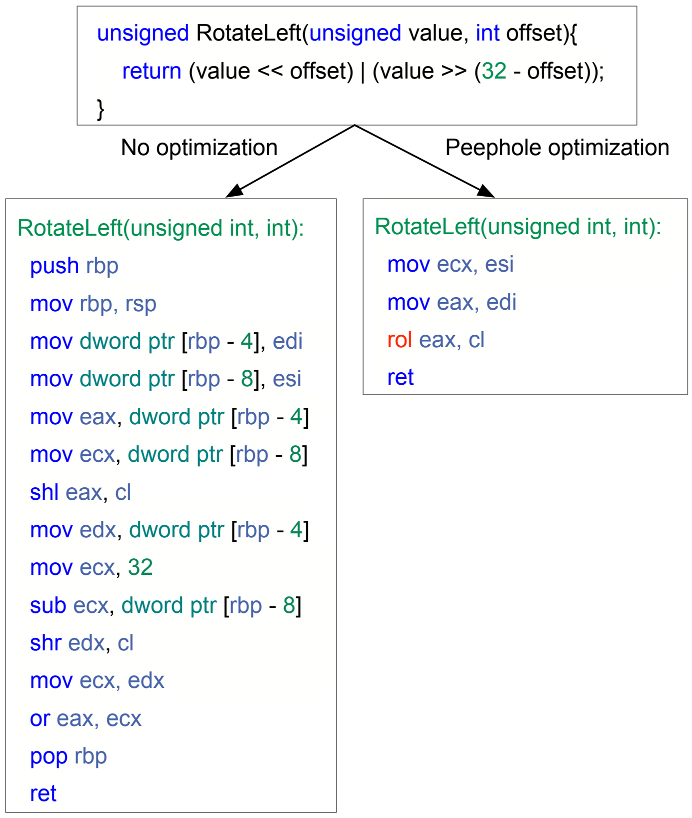
As compiler technology continues to progress, the optimization capabilities of compilers are becoming increasingly powerful. For example, LLVM's peephole optimizations have expanded to cover more than a thousand distinct types. Mainstream compilers enhance code performance and reduce binary size through peephole optimizations involving proprietary instructions. The impact of peephole optimizations on code is an intriguing topic, and we will explore their influence through two illustrative examples.

Fig. 3.1 and Fig. 3.2 present two motivating examples of peephole optimization: the optimized versions exhibit shorter instruction sequences while also showcasing significant performance improvements. Notably, the optimized code shown in Fig. 3.2 replaces the complex implementation of "counting the number of bits set to 1" with a specific instruction, `popcnt`, which was introduced since Intel Nehalem micro-architecture. As a result, the original CFG, which involves several basic blocks and a loop, is optimized as a single basic block consisting of only two instructions. Fig. 3.1 and Fig. 3.2 also reflect how peephole optimization modifies binary code at the intra-procedural level in two major ways. First, as shown in Fig. 3.1, the effect happens within a basic block: it rewrites the original basic block with a shorter instruction sequence. Second, as shown in Fig. 3.2, the effect spreads across multiple basic blocks, causing changes to the control flow graph.

Fig. 3.1 visually illustrates the application of peephole optimization by mainstream compilers to improve the rotate-left algorithm through the implementation of the ROL (Rotate Left) instruction. The ROL instruction efficiently executes a left-shift operation while preserving the leftmost bit by relocating it to the right. This optimization replaces the original sequence of instructions with a specialized one, leading to substantial improvements in program performance and a marked reduction in program size. Instead of executing numerous assembly instructions, only a single ROL instruction is needed. Moreover, related instructions such as ROR (rotate right), RCL (rotate through carry left), and RCR (rotate through carry right) can significantly boost performance and reduce code size. Concurrently, this peephole optimization entirely alters the basic block within a function, a scenario known as "1-to-1" modification at the basic block level.

Fig. 3.2 showcases the POPCNT function, a bit-counting operation that calculates the number of set bits (bits with a value of 1) in a given input. The unoptimized instruction sequence is depicted on the left, while the POPCNT instruction, applied following peephole optimization, operates on integer operands and stores the count of set bits in a destination register on the right. This optimization streamlines code size and improves runtime performance. Simultaneously, these optimizations also consolidate multiple basic blocks into a single block within a function, a scenario referred to as "m-to-1" modification at the basic block level.

Peephole optimizations play a crucial role in significantly modifying the instruction sequences within basic blocks, leading to the effective transformation of a basic block's structure. Furthermore, these optimizations also can result in the merging of multiple unoptimized basic blocks into a single, newly optimized basic block. Consequently, this process impacts the code representations of basic blocks in binary analysis methods, causing comparison methods to face challenges when dealing with



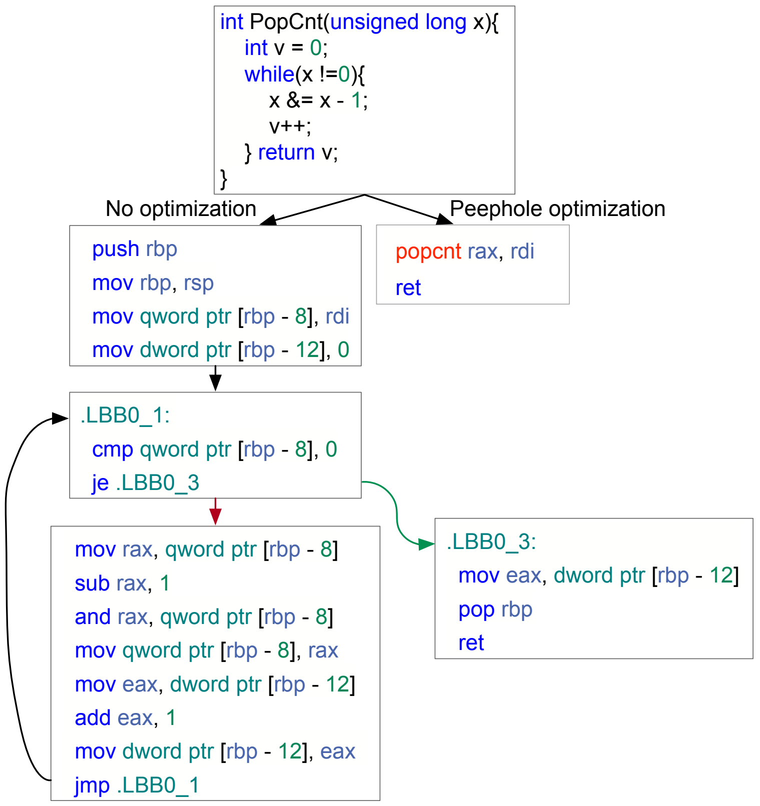
```
unsigned RotateLeft(unsigned value, int offset){
    return (value << offset) | (value >> (32 - offset));
}
```

```
RotateLeft(unsigned int, int):
  push rbp
  mov rbp, rsp
  mov dword ptr [rbp - 4], edi
  mov dword ptr [rbp - 8], esi
  mov eax, dword ptr [rbp - 4]
  mov ecx, dword ptr [rbp - 8]
  shl eax, cl
  mov edx, dword ptr [rbp - 4]
  mov ecx, 32
  sub ecx, dword ptr [rbp - 8]
  shr edx, cl
  mov ecx, edx
  or eax, ecx
  pop rbp
  ret
```

```
RotateLeft(unsigned int, int):
  mov ecx, esi
  mov eax, edi
  rol eax, cl
  ret
```

Figure 3.1: Peephole optimization with ROL instruction. The ROL instruction is utilized to implement the rotate left algorithm. Peephole optimization alters instruction sequences within basic blocks.

the altered “1-to-1” in Fig. 3.1 and “m-to-1” in Fig. 3.2 basic block matching scenarios. In summary, a basic block may undergo an extensive transformation into a new basic block due to peephole optimizations. This change can entail the removal or replacement of numerous instruction sequences, or even the amalgamation of several basic blocks into a new, optimized basic block.



```
int PopCnt(unsigned long x){
    int v = 0;
    while(x !=0){
        x &= x - 1;
        v++;
    } return v;
}
```

```
    push rbp
    mov rbp, rsp
    mov qword ptr [rbp - 8], rdi
    mov dword ptr [rbp - 12], 0
.LBB0_1:
    cmp qword ptr [rbp - 8], 0
    je .LBB0_3
    mov rax, qword ptr [rbp - 8]
    sub rax, 1
    and rax, qword ptr [rbp - 8]
    mov qword ptr [rbp - 8], rax
    mov eax, dword ptr [rbp - 12]
    add eax, 1
    mov dword ptr [rbp - 12], eax
    jmp .LBB0_1
.LBB0_3:
    mov eax, dword ptr [rbp - 12]
    pop rbp
    ret
```

```
    popcnt rax, rdi
    ret
```

Figure 3.2: Peephole optimization with POPCNT instruction. The POPCNT instruction is employed to count the number of 1 bits in a given input. Peephole optimizations alter instruction sequences and basic blocks within functions.

3.3 Current Limitations of Binary Comparison Tools of Handling Peephole Optimizations

The crux of a binary diffing approach hinges on defining a semantics-aware code representation that allows similar programs to exhibit closely related representations. At the syntactic level, the majority of research papers concentrate on discernible binary code structures for code representations, encompassing functions, basic blocks, loops, traces, control flow graphs (CFGs), and call graphs (CGs). The precision in detecting these representations relies on accurately identifying their scope within the code. At the semantics level, symbolic execution involves expressing the input-output relationships of basic blocks as formulas and employing theorem provers to verify their equivalence [22, 25, 47, 48, 49].

3.3.1 Syntactic-Level Limitations

Binary comparison methods rely on examining the Control Flow Graph (CFG) and following the basic blocks of each function in order to compare the binary code of different executables. Specifically, the binary diffing tool drills down to the same or similar function in different binary versions to compare pairs of basic blocks in order to determine their similarities. Basic blocks are sequences of instructions that execute sequentially and have a single entry and a single exit point. Binary comparison tools use the comparison of basic blocks to determine whether the functions have the same functionality or not. However, when basic blocks are optimized using peephole optimization techniques, binary comparison tools may not be able to compare them effectively. These optimizations can alter the instruction sequence of basic blocks, making it difficult for binary comparison tools to compare them at the syntactic level. As a result, most comparison tools can only perform heuristic comparisons through statistical instruction rules combined with control flow and other methods.

3.3.2 Semantic-Level Limitations

From a semantic standpoint, binary diffing tools can assess the semantics of basic blocks by utilizing resource-intensive symbolic execution techniques to address the "1-to-1" basic block transformation cases. Symbolic execution involves expressing the input-output relationships of basic blocks as formulas and employing theorem provers to verify their equivalence [22, 25, 47, 48, 49]. However, this approach is limited in addressing “m-to-1” situations, as it focuses exclusively on the internals of individual basic blocks and does not consider modifications involving multiple basic blocks.

Furthermore, implementing symbolic execution in large-scale programs presents a significant challenge due to the considerable computational overhead associated with this method. Consequently, alternative strategies must be explored to effectively manage both “1-to-1” and “m-to-1” basic block transformation scenarios in binary analysis.

It is important to be aware of the limitations of these methods in order to make more informed analyses of the basic blocks of different binaries and take appropriate actions to improve the accuracy of the analysis results. These tools can still be valuable, but they should be used in conjunction with other analysis methods to increase the accuracy of their findings.

CHAPTER 4

Related Work

In recent years, compiler optimization has garnered attention among researchers, particularly concerning its impact on binary differences. Notably, recent publications in the fields of software engineering and programming languages have emphasized the importance of understanding the various factors contributing to binary disparities during the compilation process [33, 35].

4.1 Compiler Optimization as the Main Factor Causing Syntactic Differences

A comprehensive study conducted by Kim et al. [35] systematically investigates the factors influencing binary disparities during the compilation process, while simultaneously focusing on the development of cross-architecture Binary Code Similarity Analysis (BCSA). The factors under consideration encompass the type and version of the compiler, the target architecture of the executable file, and the level of compiler optimization, among others. Through rigorous experimentation, the findings of this study reveal that the target architecture may not be the predominant factor driving BCSA. Instead, compiler optimization emerges as the most crucial element in determining the relative disparities between semantic features within the binary code.

4.2 Study Analyzing the Impact of All Available Compiler Optimization Options on Binaries

In the programming languages community, a study focuses on studying the effects of compiler optimization on binary code differences and investigating the latent capability of optimization options [36]. The research is motivated by the fact that many performance-critical applications use non-default optimization settings, and adversaries like software plagiarists or malware developers might not restrict themselves to default -Ox settings. The authors developed an auto-tuning platform called BinTuner [36], which uses a genetic algorithm to guide optimization space exploration and maximize binary code differences.

The key step in BinTuner is to design a fitness function that evaluates compilation results and steers the search process toward optimal solutions. The authors use normalized compression distance (NCD) as a fitness function to measure binary code structural differences. They tested BinTuner on various benchmarks and found that it can find custom optimization sequences that outperform default settings in all cases. Comparative evaluations with prominent binary diffing tools indicate a significant decrease in their accuracy when applied to the tuned binary code generated by BinTuner. The results highlight a new potential threat: cybercriminals can use iterative compilation to automatically generate numerous metamorphic samples. This study aims to inspire the research community to redesign resilience evaluations for binary diffing approaches and consider the impact of compiler optimization on binary code differences.

This study discusses the effects of compiler optimization on syntactic binary code representations, which is crucial for designing robust binary diffing tools. Compiler optimization algorithms can break the integrity of common code representations like binary functions, basic blocks, and control flow graphs.

Function scope is mainly affected by inter-procedural optimizations like function inlining and tail call optimization. Function inlining replaces function call instructions with the actual code of the callee function, while tail call optimization switches to a jump instruction instead of using a traditional call instruction. Both optimizations complicate function recognition and can mislead function matchings in binary diffing tools.

Basic block identification is simpler than recovering binary function scope and parameters. However, intra-procedural optimizations like loop unrolling, compound conditionals, and basic-block merging tend to produce branchless code. This can merge several basic blocks into one, violating the assumptions of basic-block-centric comparison models and requiring heavyweight inter-basic-block control flow analysis.

4.3 Study on the Effect of Function Inlining Optimization on Binaries

Moreover, the majority of binary function similarity approaches tend to disregard the influence of function inlining optimization, which involves integrating callee functions into the caller's body. In the domain of software engineering, Jia et al.'s investigation [37] the impact of function inlining on binary similarity analysis for the first time. The authors construct four datasets and propose an automatic identification method to analyze function inlining, which is found to be present in 36%-70% of binary functions under high optimization. This mismatch between the actual function inlining and the “1-to-1” mapping assumption in most binary code similarity analyses results in a 30% drop in code search performance and a 40% drop in vulnerability detection performance. Moreover, intrinsic functions are often ignored in open-source software reuse detection and patch presence testing.

Over 90% of development organizations rely on open-source components, but their inappropriate use can lead to legal and security risks. Binary code similarity analysis aims to match query binary functions with target functions. However, function inlining complicates this process, making it a “1-to-n” or “n-vs-n” problem rather than a simple 1-to-1 mapping. This challenge has not been systematically studied in previous research.

Recent studies have raised concerns about the accuracy of binary diffing tools in detecting syntactic differences due to compiler optimization. Therefore, ongoing research is necessary to enhance our understanding of how compiler optimization influences binary code discrepancies.

CHAPTER 5

Investigation on How Peephole Optimizations Affect Binary Diffing

Despite the progress achieved in binary diffing, challenges remain due to the inherently complex and low-level nature of binary code. As a result, identifying similarities or differences among programs is far from straightforward, calling for continued research and innovation in the development of code representations and similarity metrics [7, 9]. In this section, we focus on what manner peephole optimization influences the syntactic disparities in binary code. Acquiring a profound comprehension of these ramifications is essential for devising a resilient binary diffing instrument. Our investigation proceeds in two main parts. Firstly, we conducted a meticulous examination of the prevalent characteristics and patterns in peephole optimization and proposed a classification designed to impact binary differences through peephole optimization. Subsequently, we explain two examples of peephole optimization that encompass distinct types.

5.1 Peephole Optimizations Optimizing Binary Code at Intra-procedural Level

The inquiry commences by conducting a comprehensive evaluation of the inherent attributes and regularities pertaining to the realm of peephole optimization. This localized optimization approach refines code through the meticulous examination of minute segments within a program, commonly referred to as “peepholes”. We devise a classification scheme that encompasses the vast array of peephole optimizations relevant to binary code discrepancies. This taxonomical framework is crucial for clarifying the influence of optimization processes on basic block code representations, thus facilitating the progression of more proficient binary diffing methodologies to detect and address intraprocedural variances in binary code.

Peephole optimizations comprise techniques such as constant folding, algebraic simplification, strength reduction, dead code, and common subexpression elimination, along with instruction replacement that leverages hardware characteristics. These fundamental peephole optimizations ultimately result in instruction replacements within binary basic blocks, necessitating further investigation into the ramifications of these substitutions. This entails not only the substitution, removal, and repositioning of instructions within a basic block but also the merging of multiple basic blocks.

To examine the influence of peephole optimization on binary discrepancies, we consider its core optimization principle, which involves the replacement of the instruction sequence within a basic block or the merging of multiple basic blocks. Consequently, our attention is centered on the code representation wherein the basic block serves as the comparative unit. This encompasses the resilience of numerous comparison algorithms exemplified by basic block code representations.

5.1.1 Effect Within a Basic Block

Accurately identifying code representation at the block level is vital for effective comparisons in many binary differencing methods, as it allows for fine-grained analysis of code similarities between basic blocks [33]. However, this assumption becomes less reliable in real-world scenarios involving peephole optimization. For instance, peephole optimization transforms the original basic block by replacing instruction sequences, as illustrated in Fig. 5.2 (a), where the `rol` instruction optimizes the rotate-left algorithm. This optimization significantly enhances program performance

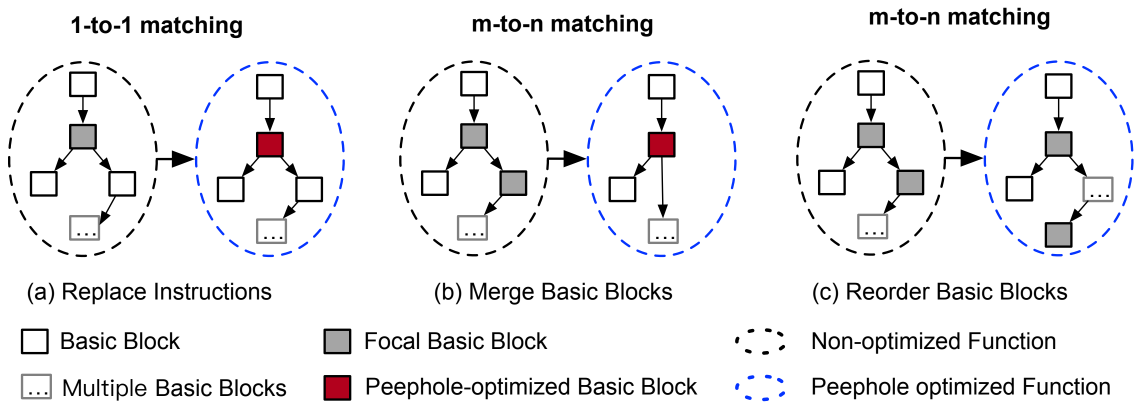
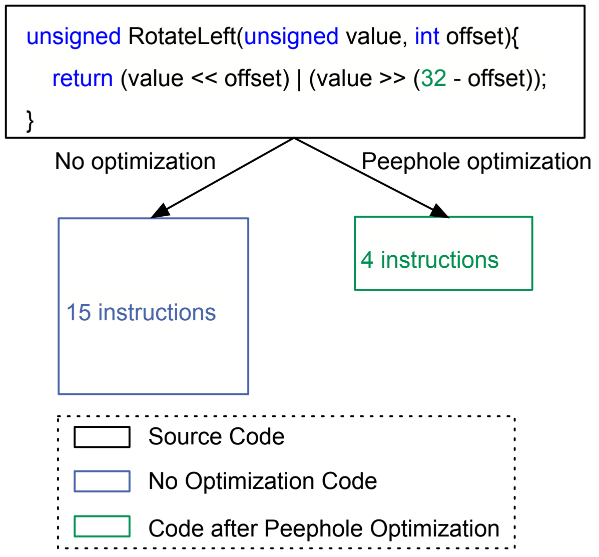


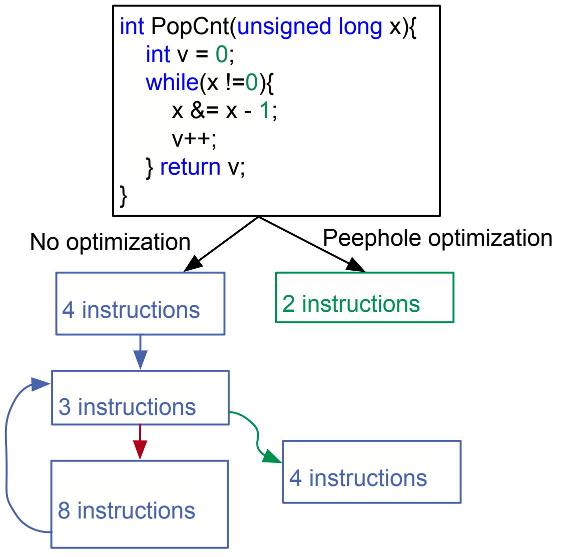
Figure 5.1: The illustration elucidates the implications of the peephole optimization process on a basic block and its subsequent influence on the control flow graph. As evidenced in (a), the effect manifests within a basic block, resulting in a modification of the original basic block by shortening its instruction sequence. As visible in (b) and (c), the effect spreads across multiple basic blocks, thus causing changes to the control flow graph.

and considerably reduces program size. We refer to this type of peephole optimization as intra-basic block peephole optimization.

As a result, industry-standard binary comparison tools (such as BinDiff [60,61]) produce inaccurate results when comparing peephole-optimized basic blocks with their original counterparts, as the optimization fundamentally alters the basic block, rendering them unidentical. Peephole optimization can also replace entirely different instructions, substantially modifying the instructions within the basic block. Consequently, it affects binary comparison methods that rely on instruction frequency statistics and opcodes within basic blocks. Although methods like BinHunt [62] can utilize resource-intensive symbolic execution to address peephole optimization within basic blocks and compare basic block input-output relations through accurate semantic modeling, this approach is impractical for large-scale binary search scenarios due to the overhead of symbolic execution. Therefore, the basic-block-centric comparison model assumption is compromised by the presence of peephole-optimized code


unsigned RotateLeft(unsigned value, int offset){
return (value << offset) | (value >> (32 - offset));
}
No optimization
Peephole optimization
15 instructions
4 instructions
Source Code
No Optimization Code
Code after Peephole Optimization

(a) ROL instruction changed intra-basic block.



(b) POPCNT instruction merged inter-basic blocks.

Figure 5.2: The technique of peephole optimization has been found to exert a significant impact on various categories of basic blocks.

within basic blocks. These assumptions are based on a simplistic "1-to-1" matching paradigm, where basic blocks extracted from two distinct binary files undergo pairwise comparison [16, 22, 47, 55, 62], or multiple one-to-one cases form an "n-to-n" matching scenario [63], in which several basic blocks from the same function are optimized during the same peephole pass, leading to modifications in multiple basic blocks.

5.1.2 Effect Across Multiple Basic Blocks

Peephole optimization not only involves modifying the instruction sequence within basic blocks, but also encompasses various operations such as deletion or merging of multiple basic blocks, and modifications to the control flow graph between basic blocks. An example of this is the `popcnt` optimization illustrated in Fig. 5.2 (b). The `popcnt` function is a bit-counting operation that counts the number of set bits (bits with a value of “1”) in a given input. In Fig. 5.2 (b), the unoptimized instruction sequence is shown on the left, while the peephole-optimized instruction sequence on the right utilizes the `popcnt` instruction directly to operate on integer operands and stores the count of set bits in a destination register. This optimization reduces code size and improves runtime performance. Notably, this optimization and modification involve multiple basic blocks. We refer to this type of peephole optimization as inter-basic block peephole optimization.

When applying this type of peephole optimization technique, modifications to basic blocks involve multiple blocks, leading to a mismatch between un-optimized and optimized basic block pairs. Consequently, heuristic matching based on basic block pairs, such as BinDiff, may fail, reducing the accuracy of the final result. Additionally, methods like BinHunt [62] focus solely on symbolic execution analysis for individual basic blocks, without taking into account simultaneous changes spanning multiple basic blocks within a function. In other words, when the original m basic blocks are merged into n new basic blocks, these methods would fail to match in the “m-to-n” scenarios.

Furthermore, optimization patterns for complex “m-to-n” matching scenarios involving multiple basic blocks may impact AI-based binary comparison methods that rely on basic block embeddings. This is because the embedded vector values corresponding to basic blocks have been substantially modified, thereby undermining the assumption of a basic-block-embedding comparison model.

Although peephole optimization can improve the performance and efficiency of code, its implementation of changed basic blocks can compromise binary code analysis and comparison. To address the issue of modifications to multiple basic blocks, it is necessary to conduct an in-depth inter-block control flow analysis. This involves analyzing the control flow graph to identify the relationships between basic blocks and their respective instructions. By understanding the inter-block control flow, it is possible to match pre-optimized and optimized basic block pairs accurately, enabling binary matching algorithms to produce more precise results.

5.2 Answer to RQ1

Peephole optimization affects binary code at the intra-procedural level in two distinct ways. Firstly, it modifies the original basic block by shortening its instruction sequence, thus resulting in an effect that is confined within the block. Secondly, the effect spreads across multiple basic blocks, ultimately altering the control flow graph. This introduces complications in the pair-wise comparison of basic blocks and poses a challenge to binary diffing tools that rely heavily on such comparisons as their fundamental framework. In Chapter 8, we will investigate the extent to which peephole optimization occurs throughout the entire optimization process.

CHAPTER 6

Approach

It is crucial to highlight that LLVM does not allow users to selectively disable or enable peephole optimizations through specific options such as (-fno-peephole or -disable-instcombine). Consequently, one cannot enable peephole optimizations in an unoptimized setting (O0) or activate all default optimizations (O3) while solely disabling peephole optimizations and retaining other optimizations. In essence, users are unable to compile distinct binary codes containing only peephole optimizations or no peephole optimizations by setting the disabling or enabling of peephole optimizations during compilation. Thus, assessing the impact of peephole optimizations on binary code remains an open technical challenge.

Once compiled, the source code undergoes the cumulative effects of all optimizations, resulting in the final binary code. This makes it impossible to isolate the influence of a single optimization, such as peephole optimization, at the binary level. Thus, our focus lies on the compilation's optimization process, analyzing the intermediate representation (IR) to extract the pass-affected peephole optimization from the compilation process.

The peephole optimization's scope is function-based. To accurately analyze the IR comparison of a specific function before and after optimization, we employ the LLVM translation verification tool. This ensures that the IR semantics of the two function versions being compared are identical, and peephole optimizations solely modify the function's syntax. We utilize Alive2 [45] to capture each optimization's code translation process in the optimization pipeline and extract the intermediate code representation before and after peephole optimization for each function. This customized tool allows us to confirm the correctness of each compared function pair (before and after peephole optimization) and effectively isolate and evaluate the specific effects of peephole optimization separately from the overall optimization process.

6.1 Overview

Our workflow is illustrated in Fig. 6.1. Initially, we capture a snapshot of the peephole optimization pipeline during the optimization process, preserving the intermediate representation (IR) before and after optimization. Subsequently, we perform an IR comparison of the peephole optimization pipeline (blue for pre-peephole optimization IR and red for post-peephole optimization IR) and confirm that the compared pair shares the same semantics. Lastly, we conduct syntax difference analysis on the two IR versions with identical semantics but different syntax to identify the modifications made to the basic blocks and instructions within the function under the current peephole optimization pass.

It should be noted that in the optimization process, the optimization manager may call each optimization multiple times, and the peephole optimization technique may optimize the same function multiple times. Consequently, the optimization process involves the application of peephole optimization multiple times, either for the same function or different functions, and these optimizations are intermixed throughout the process.

Given that peephole optimization is performed at the function level, multiple calls may indicate repeated optimization of the same function or separate optimizations of multiple functions. For example, Figure 2.1 illustrates multiple occurrences of peephole optimization, which may be optimizations of different functions or multiple optimizations of the same function. Therefore, it is crucial to identify and analyze

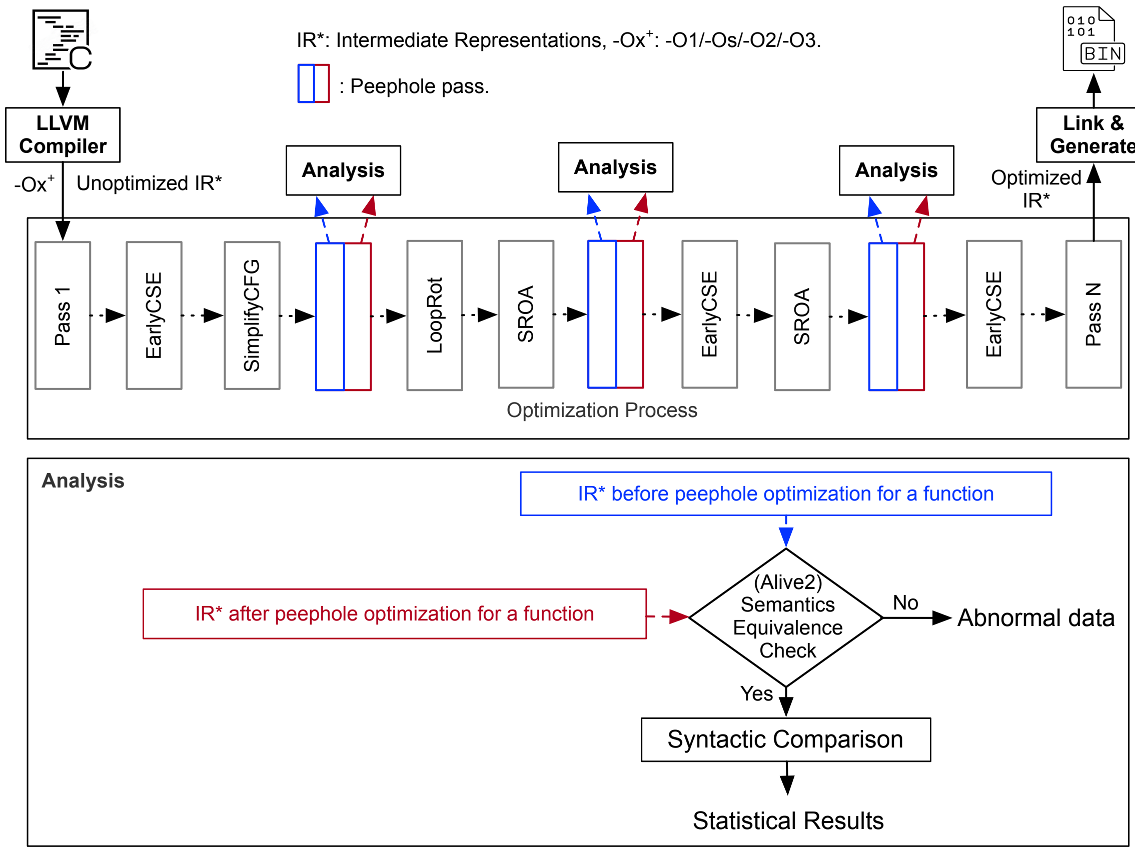


Figure 6.1: Workflow for analyzing peephole optimizations.

multiple peephole optimizations separately for different functions based on their functionality when analyzing peephole optimization.

6.2 Challenge: Difficult to Isolate the Effect of Peephole Optimizations at Binary Level

Given the complexity of real-world source programs, compiler optimization necessitates the execution of numerous optimization iterations, which can range from thousands to tens of thousands. Each of these iterations is referred to as a pass [1, 2]. Operating sequentially, these passes target all functions within the program to ensure a comprehensive optimization process.

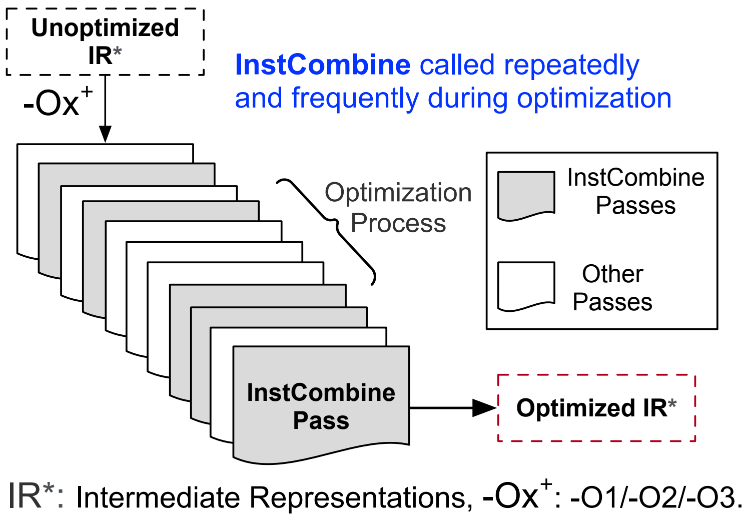

Unoptimized IR*
-Ox+
InstCombine called repeatedly and frequently during optimization
Optimization Process
InstCombine Passes
Other Passes
InstCombine Pass
Optimized IR*
IR*: Intermediate Representations, -Ox+: -O1/-O2/-O3.

Figure 6.2: In the context of compiler optimization, the process is characterized by a complex, iterative nature. Each function within the source program undergoes multiple, distinct optimizations, and may repeatedly invoke the same optimization following others, in pursuit of any possible opportunity to enhance the optimized IR, such as the peephole optimization (implemented by LLVM's InstCombine pass [3, 4]) called frequently during optimization.

The initial phase of this intricate procedure involves transforming the source code into an intermediate representation (IR). This IR serves as a platform-independent, low-level format that facilitates further optimization. Crucially, this transformation bridges the gap between the high-level source code and the target machine code, promoting a more efficient and streamlined optimization process. Subsequent to the generation of the unoptimized IR, optimization passes are iteratively applied. These passes encompass a diverse array of techniques, including constant propagation, dead code elimination, and loop invariant code motion, among others. The primary objective of these optimizations is to enhance the efficiency and performance of the target program while preserving its original semantics.

From the perspective of compiler optimization, the optimization process is akin to an assembly line team, with each member stationed along the line to perform consecutive optimizations on the parsing code. The code is then handed over to the next member for further processing. After several iterations, the team's work concludes when the code has been optimized to the greatest extent possible, and the optimized IR is delivered to the linker. For example, the peephole optimization (implemented by the InstCombine pass in LLVM) is repeatedly called by the optimization manager in Fig. 6.2 with a gray highlighted background. Through numerous iterations of the Peephole optimization technique, a particular function is subject to continuous refinement to achieve the overarching objectives of minimizing program size, conserving resource utilization, and enhancing overall execution performance.

Ultimately, the constructed binary executable file loses the specific details of each optimization process. As a result, all optimization outcomes are combined into a single binary executable file. Consequently, attempting to identify the effect of a specific optimization, such as peephole optimization, solely from the binary level is nearly unfeasible.

6.2.1 Solution

To address this challenge, we focus on the changes in the Intermediate Representation (IR) during compilation, rather than on the impact of final peephole optimizations on the binary code. Fig. 6.2 illustrates how we accomplish this by concentrating on the *InstCombine Pass* [3, 4] in the IR during compilation.

In the realm of optimization analysis, examining the alterations introduced by peephole optimization to the IR can yield invaluable insights into its impact on the overall optimization process. This approach enables us to isolate the effects of peephole optimization from other optimizations and gain a deeper understanding of its contributions to the optimization process.

Furthermore, quantifying the influence of peephole optimization on the program being optimized can be achieved by counting and analyzing these changes. Scrutinizing the changes induced by peephole optimization to the IR and quantifying its influence on the program being optimized can provide valuable insights into the overall optimization process's contributions. While not a perfect solution, this method offers a beneficial starting point for assessing peephole optimization's impact and guiding subsequent optimization endeavors.

6.3 Challenge: How to Isolate the Effects of Peephole Optimization from Complex and Intertwined Optimization Processes

In our research, we delve into the effects of peephole optimization on Intermediate Representation (IR) by examining the influence of peephole optimization on each function during the LLVM optimization process. However, even when analyzing the IR optimization process, certain technical challenges persist.

Specifically, peephole optimization's scope of impact encompasses the function level. Consequently, we endeavor to identify a suitable approach that enables sequential analysis of all functions within the program, commencing with a single function and extending to all others. The goal of this method is to capture the multiple code modifications elicited by peephole optimization on each function, thereby offering a comprehensive understanding of its effects on the optimization process.

6.3.1 Multiple Modifications in the Same Function Induced by Peephole Optimization During the Optimization Processes

We have observed that peephole optimization often applies multiple modifications to the same function during the optimization process. This iterative procedure is typically carried out in several stages, with each stage focusing on specific aspects of the code being optimized. As depicted in Fig. 6.3, the peephole optimization process involves multiple code modifications for a single function. The highlighted regions in red, green, and blue represent different functions that undergo optimization. For instance, the function highlighted in red experiences modifications by peephole optimization at four distinct locations, namely (a), (b), (c), and (d). Similarly, the function highlighted in green undergoes modifications by peephole optimization at positions (e) and (f).

After the function has been subjected to other optimizations, such as loop optimization, the peephole pass is invoked once again to assess the potential for further peephole optimization. This indicates that peephole optimization involves numerous code modifications for a single function throughout the entire optimization process.

Peephole optimization implements repeated local instruction sequence modifications during the optimization process. These modifications contribute to the optimization of the function in the process. Consequently, it is crucial to devise an appropriate method for tracking each function and recording the modifications applied by peephole optimization during each iteration.

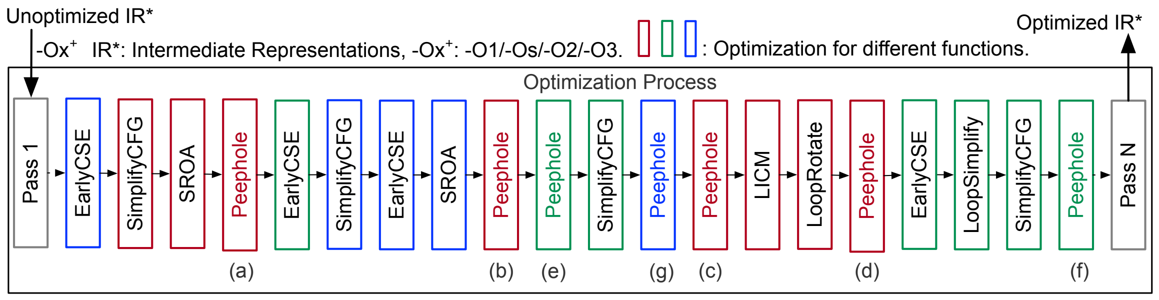


Figure 6.3: The optimization of multiple functions is interleaved and mixed together throughout the optimization process. *EarlyCSE*: identifies and removes repetitive subexpressions or operations within a function. *SimplifyCFG*: streamlines the program's Control Flow Graph (CFG) by eliminating redundant control structures, merging adjacent basic blocks, and removing unreachable code. *LoopRotate*: by rotating the loop body, it aligns the loop header with the most frequent exit path, reducing branches during execution, and improving code locality and cache utilization. *SROA* (Scalar Replacement of Aggregates): Enhances performance by breaking down aggregate data structures like arrays and structures into individual scalar variables. This promotes efficient memory access and further optimization opportunities for other LLVM passes, improving execution efficiency, especially in applications with intricate data structures and memory access patterns. *LoopSimplify* pass is responsible for canonicalizing loop structures, transforming them into a more manageable and predictable form for further analysis and optimization. *LICM* (Loop Invariant Code Motion) Pass aims to optimize performance by moving loop-invariant code out of the loop body, reducing redundant computations.

6.3.2 Interwoven Optimization Processes Across Distinct Functions

We have observed that peephole optimization typically interweaves the optimization processes of different functions. The iterative optimization of various functions is often blended, with each phase concentrating on specific aspects of the code being optimized.

As illustrated in Fig. 6.3, the peephole optimization process encompasses multiple code modifications for numerous functions, intertwined in their execution. The regions highlighted in red, green, and blue represent distinct functions that undergo optimization. For instance, during this optimization process, the blue function is initially optimized through the identification and removal of repetitive subexpressions or operations within the function. Subsequently, the red function is optimized by eliminating redundant control structures, performing Scalar Replacement of Aggregates, and applying peephole optimization. Then, the green function is optimized and continues like this until the end.

The optimization passes for different functions in peephole optimization are interlaced, resembling parallel processing pipelines. They do not influence one another while concurrently optimizing various aspects of distinct functions. These modifications collectively contribute to the overall success of peephole optimization. Therefore, to find out the peephole optimization specific to each function from the intertwined optimization passes, it is crucial to devise an appropriate method to track each function and record the modifications applied by the peephole optimization during each iteration.

6.3.3 Interaction between Peephole Optimizations and Preceding Interleaved Optimizations in Optimization Processes

In our study, we aim to identify peephole optimizations for each function within interleaved passes. One approach is to track each function and record any modifications made by peephole optimizations throughout the optimization process. However, advanced optimizations like function inlining can disrupt this tracking process by merging multiple functions, while loop optimizations can alter the order and number of basic blocks, leading to the disappearance of functions or blocks during tracking.

The intricate relationship between optimizations demonstrates a complex interdependence and interconnectedness that must be carefully considered in the compiler optimization analysis process. Peephole optimization serves as a step within the optimization pipeline, relying on the outcomes of previous optimizations and influencing subsequent optimizations in turn. Therefore, it is crucial to account for the impact of various optimizations on the overall optimization process.

As shown in Fig 6.3, the red highlighted function undergoes peephole optimization at position (d). However, prior to this step, the function has undergone Loop Invariant Code Motion optimization, followed by rotating the loop body to align the loop header with the most frequent exit path. These optimizations improve code locality and enhance cache utilization. These optimizations may lead to changes in the loop structure within the function, resulting in alterations to the number and order of basic blocks. These changes create more opportunities for peephole optimization, but may also cause the instruction sequence previously optimized by peephole optimization at position (c) to be removed or replaced.

If we were to systematically track the entire optimization process, documenting every iteration of optimization for each function from start to finish, and then extract the corresponding effects of peephole optimization, we would encounter two problems. Firstly, other optimizations that merge functions and basic blocks may cause truncation of optimization records for specific functions during the tracking process. This may result in incomplete information about the optimization paths, as short tracking of optimization paths may not capture the complete picture. Secondly, we must deal with the cost of analyzing a vast number of optimization processes for practical programs that may require more than ten thousand optimization steps. For instance, the bzip2.c program has over 16,000 passes. If we consider four optimization levels (O1, Os, O2, and O3), we would need to track a total of 4 * 16,000 passes. This would require a significant amount of time and resources, and the resulting data may be incomplete.

In conclusion, while tracking and documenting all changes made by optimizations in the pipeline may provide a comprehensive view of the process, it comes with significant costs and may be obstructed by the merging of functions during other optimizations. As a result, it may be difficult to obtain a clear understanding of the impact of peephole optimization.

6.3.4 Our Solution

In fact, a more targeted approach to analyzing the impact of optimizations in the optimization process may be more effective than tracking and documenting all optimizations. One approach is to focus specifically on the peephole optimization pass, treating each peephole optimization pass as a separate optimization rather than continuously tracking all steps of peephole optimization for a given function. We can analyze each peephole pass separately, comparing the IR before and after optimization at the function level (since the scope of peephole optimization is at the function level), and examining the impact of each peephole pass on the basic blocks within the function.

In our study, for example, we do not track all optimized passes for the red, green, and blue functions in Fig. 6.3. Instead, we will analyze the specific impact of each peephole optimization on different functions individually. This involves separately analyzing the positions (a), (b), (c), and (d) in the function marked in red 4 times, and recording the influence of each peephole optimization in this pass on the basic block of the function. Finally, we will add the results of the 4 times together and obtain an average contribution rate to the impact of peephole optimization on the current program. Although this method is not perfect, it can help us evaluate the utilization of peephole optimization and introduce complications in the basic block comparison methods when the overhead is acceptable.

By counting the impact of each peephole optimization pass on a specific function and adding up all the impacts of peephole optimization for that function, we can obtain statistical results on the specific function's impact, including the peephole optimization's final impact on all functions in the program and its average contribution rate at different optimization levels. This method of analyzing the impact of peephole optimization by separately analyzing each peephole pass, then summing up and averaging the results, can provide a clear understanding of the contribution of peephole optimization as a local optimization in the entire optimization process.

CHAPTER 7

Experiments and Datasets

In this chapter, we present a thorough overview of the experimental design, setup, and data sets utilized in our study. This comprehensive account ensures that both distinct experiments are clearly and effectively conveyed.

7.1 Experimental Design

In our research, we investigate the impact of peephole optimization on binary code through two distinct experiments.

Firstly, in Chapter 8, we analyze the extent to which peephole optimization occurs during various compiler optimization levels by capturing the peephole optimization process that takes place during LLVM compilation. This analysis is conducted from three different perspectives to determine whether peephole optimization plays a dominant role in the compiler process.

Secondly, in Chapter 9, to address the challenge of hard-to-isolating peephole optimization for individual analysis in binary segments, we collect hundreds of program units with peephole optimization properties from the experiment as a new dataset, covering two different types of peephole optimizations. We employ two representative binary comparison tools to examine the impact of peephole optimization on these two distinct binary comparison techniques. We assess two different categories of datasets, each influenced by peephole optimizations related to binary code generation in varying ways. By binary diffing these two datasets separately, our goal is to understand the effect of peephole optimization on binary difference methods and their effectiveness in various situations.

7.2 Experimental Setup

Our experimental testbed is comprised of twelve i5-10400 CPUs operating at a frequency of 2.90GHz, along with 32GB of RAM, and runs on the Ubuntu 22.04 LTS operating system.

In our analysis of peephole optimization experiments, we observed that in real programs, the pass responsible for peephole optimization and control flow graph optimization within the optimization pipeline is invoked multiple times, often reaching thousands of iterations. However, it is important to note that not every invocation of these optimization passes necessarily modifies the Intermediate Representation (IR) code. Instead, these invocations can be seen as attempts to optimize the code. Consequently, for the first experiment in Chapter 8, we carefully designed our approach to exclude passes that were called but did not result in any code changes. This strategy ensures that the experimental results encompass only those peephole passes that genuinely modified the code.

In the first experiment, we compile all test set programs using a total of four optimization levels: O1, O2, Os, and O3. Our program tracks the compilation process for each individual program. Specifically, during the compilation process, we capture the function-level optimization changes (changes to basic blocks before and after optimization) from the peephole pass snapshots. This allows us to perform cumulative statistics of all functions and programs.

In the second experiment, we use binary code compiled with O0 as the baseline and binary files compiled with O3 as the comparison. Due to the extracted program collection carrying only peephole optimization features, using O3 as the optimization

Table 7.1: LLVM's unit test suite.

Dataset	# of Programs	Binary Versions
PolyBench	30	150
CoyoteBench	4	20
Adobe-C++	6	30
BenchmarkGame	8	40
Dhrystone	2	10
Linpack	1	5
McGill	4	20
Misc	27	135
Misc-C++	5	25
Shootout	15	75
Shootout-C++	25	125
SmallPT	1	5
Stanford	11	55
Regression	50	250
bzip2	1	5
UnitTests	142	710

level will result in enabling peephole optimization only. We then compare the generated binary files using comparison tools to obtain results, such as the proportion of basic block match rates.

7.3 Two Different Datasets

In the first experiment, we utilize LLVM's unit test suite, featuring classic programs gathered over a decade and extensively employed in compiler testing, to examine peephole optimization's influence on binary diffing. As shown in Table 7.1, this suite encompasses hundreds of programs and various small test sets. These utilities serve as crucial research datasets due to their practical applications and representative attributes. These programs were selected as datasets due to their practical use and representativeness in their respective domains, which will help us more comprehensively analyze and evaluate the impact of peephole optimization techniques on binary code differences.

In the second experiment, the ideal method to evaluate the impact of peephole optimizations on binary code would involve directly compiling programs from the LLVM test set and analyzing the resulting binary code. Unfortunately, LLVM does not provide options for selectively enabling or disabling peephole optimizations, making it challenging to isolate their influence at the binary level. To circumvent this issue, we adopted a compromise strategy that involves statistically analyzing the test set program's optimization process and extracting common and general peephole optimizations. These small program units, characterized by specific structures or specialized peephole instructions, effectively represent a subset of peephole optimizations and are divided into two categories based on different types. Using this curated dataset, we assess the impact of peephole optimization on binary code, albeit with certain limitations. Despite not covering the entire spectrum of peephole optimizations, this approach allows us to investigate the influence of two distinct types of peephole optimizations on binary code and the performance of binary comparison tools, providing valuable insights into the effect of peephole optimizations on binary code differences.

CHAPTER 8

Investigation on the Extent of Peephole Optimization

Before delving into the extent of peephole optimization, we find that peephole optimization is not uniform across all scenarios. Firstly, peephole optimization may vary depending on the different levels of compiler optimization settings, as it is invoked multiple times for local code optimization after other optimizations have been completed. Consequently, peephole optimization is influenced by other compiler optimizations and changes according to the number of optimization options enabled. Secondly, at different compiler optimization levels, the popularity of individual optimization passes within the overall optimization process varies, as does the popularity of peephole optimization. Lastly, at different compiler optimization levels, the extent of peephole optimization's impact on basic blocks and functions also varies.

Therefore, in this section, we propose three research components to facilitate a better understanding of the scope of peephole optimization.

8.1 What Is the Extent of Peephole Optimization under Different Levels of Compiler Optimization Settings?

In our experiment, we assessed the applicability of peephole optimization across various LLVM optimization levels, ranging from O1 to O3. Each level is designed to fulfill a specific objective and encompasses a diverse array of optimization options, thereby leading to distinct effects on code generation. O1 is geared toward lightweight optimizations, enhancing execution speed with minimal impact on compilation time. O2 strikes a balance between execution speed and compilation time by employing techniques such as function inlining, loop unrolling, and constant propagation. Os, on the other hand, emphasizes code size optimization through code compression and the elimination of unused functions. O3 embodies a high-level optimization level, prioritizing maximum execution performance through advanced methods, including loop optimization, vectorization, and memory optimization. By comparing the optimized results with the unoptimized baseline (before and after peephole optimization in the InstCombine pass), we were able to gauge the influence of peephole optimization on binary code differences.

Fig. 8.1 shows the statistics of peephole optimization in the dataset compiled by LLVM Clang 13.0.0 from O1 to O3. Considering the proportion of the peephole pass invoked by the optimization manager relative to all other optimization passes, it is observed that at the O1 optimization level, the median invocation frequency of the peephole optimization pass is 32%. This frequency ranges from a minimum of 14.6% to a maximum of 39.7%, reflecting the differences in complexity among various programs.

As the optimization level progresses from O1 to O3, there is an increase in the number of enabled optimization options, which subsequently leads to a rise in the number of passes not related to peephole optimization throughout the optimization process. Consequently, with the activation of additional optimization options and the involvement of more diverse optimizations, the overall proportion of peephole optimization invocation frequency experiences a declining trend. In essence, as the number of activated optimization options and participating optimizations increases, the proportion of peephole optimization decreases.

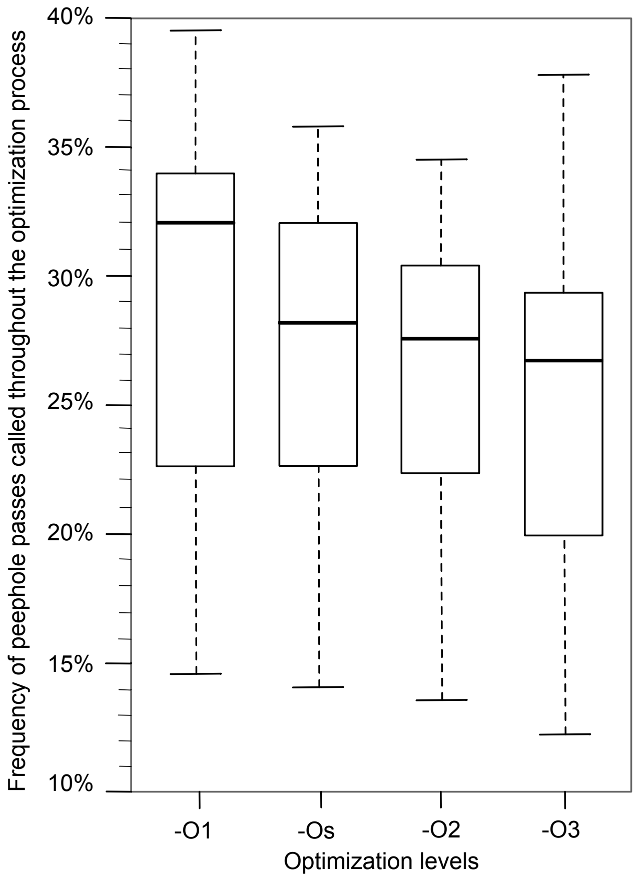


Figure 8.1: Statistics on how often peephole optimization is invoked by the optimization manager at different compiler optimization levels throughout the optimization process.

Within the O1 to O3 optimization levels, the average invocation frequency of peephole optimization varies from 19.7% to 33.9%. The minimum and maximum recorded frequencies are 11.8% and 39.7%, respectively.

8.2 What Is the Most Frequently Used Pass Ranking in the Optimization Process Across Various Optimization Levels?

During the optimization process, the optimization manager applies diverse optimizations repeatedly to enhance the code, based on the specific requirements of various functions within the program. Certain critical optimization passes frequently occur throughout the process. Therefore, we conducted a statistical analysis of the invocation of all common passes and presented the top 10 most popular optimization passes, based on their highest invocation frequency, under different optimization levels, in Fig. 8.2. The result indicates that the peephole optimization pass is the most frequently utilized among the top 10 popular optimization passes. Across the four optimization levels, the average invocation frequency by the optimization manager ranges between 25% and 29%.

8.3 What Percentage of Basic Blocks and Functions Are Affected by Peephole Optimization?

To investigate the impact of peephole optimization on the basic blocks and functions of programs in a test suite set throughout the entire optimization process, we conducted a statistical analysis of the modifications made to the basic blocks in each function under different optimization levels. As shown in Figure 8.3, at the O1 level, the median impact proportion of peephole optimization on the basic blocks in the program was 40.5%. With the increase of optimization level and the introduction of more types of optimization, the range of proportion of basic blocks affected expanded (blue box), as different program structures resulted in a greater possibility of optimization. If a program is simple, the impact of peephole optimization may be reduced, while if its complexity is suitable for peephole optimization, it can lead to a greater possibility of optimization. For functions (pink box), with the introduction of more optimization at higher levels, more opportunities were provided for peephole optimization, resulting in more functions being optimized. The average impact proportion of peephole optimization on the basic blocks in the program ranged from 32.8% to 60.2%, with a minimum impact of around 22% and a maximum impact of 76.3%. The average impact proportion of peephole optimization on the functions in the program ranged from 34.7% to 63.8%, with a minimum impact of 22.6% and a maximum impact of 79.1%.

8.4 Answer to RQ2

In summary, our findings show that peephole optimization is the most popular optimization pass, with an average invocation frequency ranging from 20.1% to 37.9%. At all O1∼O3 levels, the average percentage of basic blocks modified by peephole optimization fluctuates between 32.8% and 60.2%, with a maximum value of 76.3%.

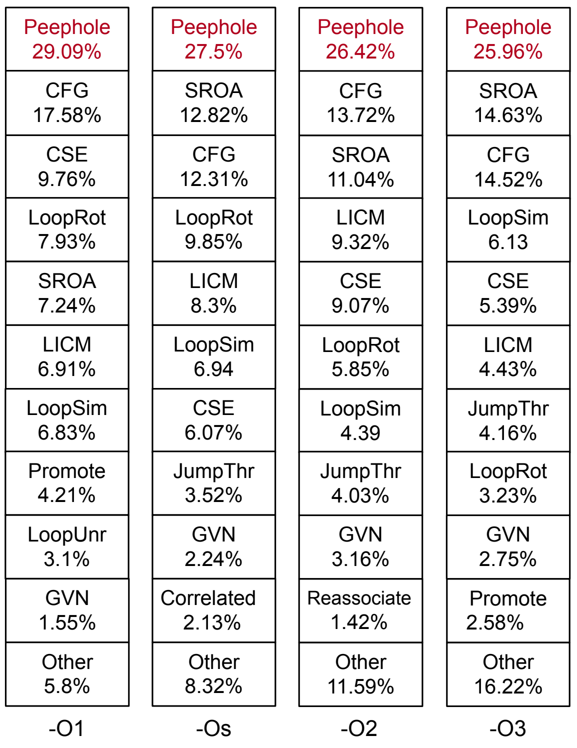


Figure 8.2: The Peephole optimization pass is highly popular among the top 10 optimization passes, with an average invocation frequency ranging between 25% and 29% across the four optimization levels. Other frequently used optimization passes include SimplyCFG, which simplifies the control flow graph by removing redundant instructions and blocks, making it simpler for subsequent passes to analyze and optimize the code. CSE (Common Subexpression Elimination) identifies common subexpressions in the code and replaces them with a single computation, reducing the number of executed instructions and improving performance. Looprotate reorders loops to improve code locality and reduce the number of branch instructions executed. SROA (Scalar Replacement of Aggregates) replaces complex data structures with simpler scalar variables, reducing memory usage and improving performance. LICM (Loop Invariant Code Motion) identifies code that is repeatedly executed in a loop and moves it outside the loop, reducing the number of executed instructions and improving performance. These optimization passes are crucial in significantly enhancing the code's performance.

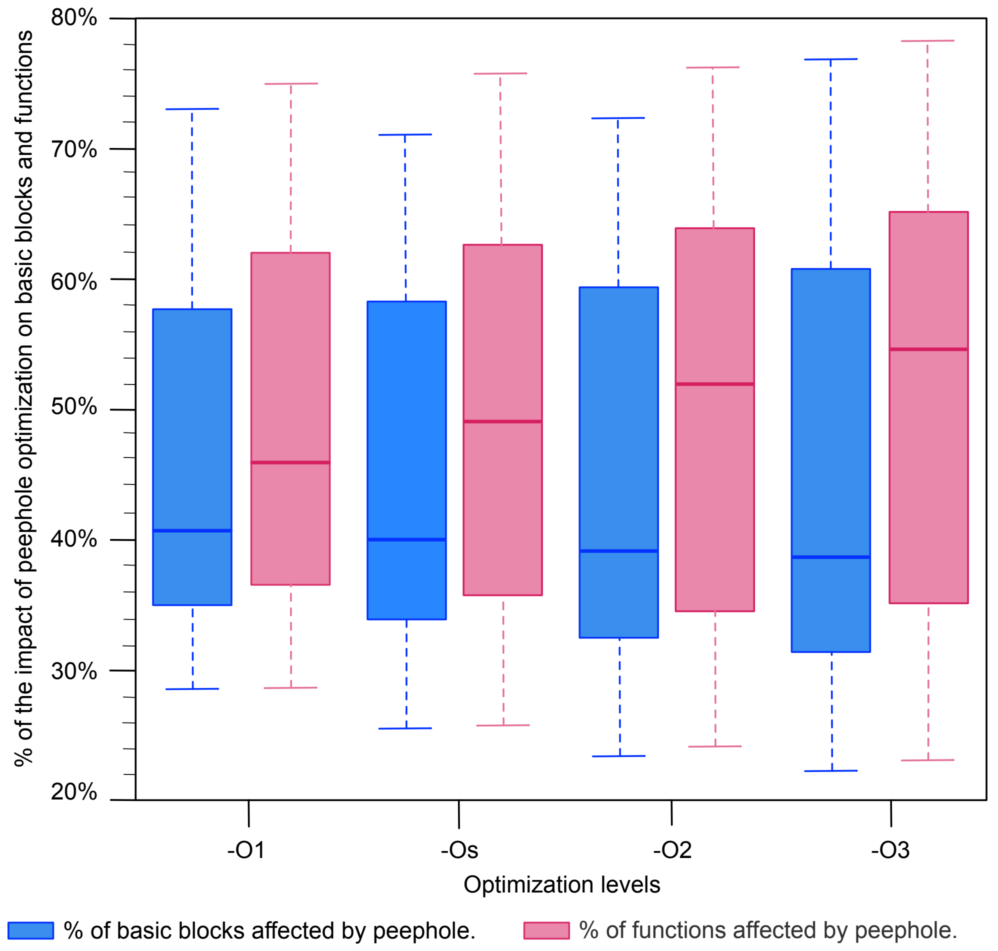


Figure 8.3: Effect of peephole optimization on binary basic blocks and binary functions.

CHAPTER 9

Investigation of the Implications of Peephole Optimization on Two Binary Diffing Methodologies

The significance of peephole optimization's impact on existing binary diffing techniques cannot be understated, particularly due to its influence on performance. In this chapter, our focus will be directed toward examining the consequences of peephole optimization on a pair of widely-used binary diffing tools, emphasizing the necessity of addressing this optimization approach in the realms of software security and reverse engineering.

Initially, we will present an in-depth description of the chosen test dataset, accentuating the diverse attributes that make it an appropriate subject for our study. Following this, we will proceed to discuss the two binary diffing tools selected for this analysis, elaborating on their distinct features, methodologies, and the reasoning that guided their inclusion in our research.

To ensure a thorough evaluation, we have carried out assessments on two separate categories of datasets, with each category being differently impacted by peephole optimization in relation to binary code generation. Through the separate examination of these categories, our objective is to offer a comprehension of the repercussions of peephole optimization on binary diffing methodologies, as well as their efficacy across a range of situations.

9.1 Extracted Two Types of Peephole Dataset

The optimal approach to assessing the impact of peephole optimizations on binary code involves directly compiling with programs from the LLVM test set and verifying the resulting binary code. By analyzing the binary code after peephole optimization and the subsequent changes in basic blocks generated by these optimizations, it would be possible to evaluate the performance of binary comparison tools. Regrettably, LLVM does not offer users the ability to selectively disable or enable peephole optimizations through options such as (-fno-peephole or -disable-instcombine). Consequently, it is not feasible to enable peephole optimizations in an unoptimized setting (O0) or to activate all default optimizations (O3) while disabling only peephole optimizations and maintaining other optimizations.

In essence, users are unable to compile distinct binary codes containing only peephole optimizations or no peephole optimizations by setting the disabling or enabling of peephole optimizations during the compilation process. As a result, determining the influence of peephole optimizations on binary code remains a formidable technical challenge. The primary obstacle lies in the inability to isolate the impact of specific peephole optimizations from these intricate programs at the binary level.

To address this issue, a compromise strategy has been adopted, which involves conducting a statistical analysis of the optimization process for the test set program and extracting some common and general peephole optimizations from all programs. These small program units, characterized by specific structures or specialized peephole instructions, can effectively represent a subset of peephole optimizations. The program units are then divided into two categories based on different types, resulting in the construction of a collection containing hundreds of refined programs of two distinct types in Table 9.1.

Table 9.1: Extracted Peephole Dataset

Two types of collected peephole optimization program units	# of Units
Type 1: Effect within a basic block	192
Type 2: Effect across multiple basic blocks	32

Utilizing this curated dataset of peephole programs, we can assess the impact of peephole optimization on binary code, albeit with certain limitations. Although the dataset may not be fully representative of the entire spectrum of peephole optimizations, it provides a pragmatic method for investigating the influence of two distinct types of peephole optimizations on binary code and the performance of binary comparison tools. By analyzing the impact of these programs on the comparison tool, we can gain valuable insights into the effect of peephole optimizations on binary code differences.

9.2 Binary Diffing Tools' Selection

In this study, two prominent binary comparison tools from the past decade have been selected for analysis, namely BinDiff [60, 61] and DeepBinDiff [64]. These tools encompass a broad range of widely employed and established binary analysis methods, as well as state-of-the-art AI-based approaches, thereby providing coverage for examining the effects of peephole optimization on basic block and instruction sequence modifications.

BinDiff, a prominent binary differencing tool in the industry, leverages IDA's disassembly code as its input source [32] and integrates a multi-layered statistical feature system, including control flow, functions, and basic blocks, to improve the efficiency of graph matching processes [60, 61]. This tool exhibits a strong capability to endure intermediate syntactic variations, such as register swapping and instruction rearrangement. In contrast, DeepBinDiff [64] adopts an unsupervised deep neural network-based approach for binary diffing, initially learning basic block embeddings through unsupervised deep learning. These embeddings represent specific blocks with semantic and contextual information extracted from the interprocedural control flow graph (ICFG). The prototype implementation indicates that these embeddings are employed to efficiently and accurately compute basic block similarities. These two renowned tools in the industry assist in understanding and analyzing the impact of peephole optimization on binary code differences. For all evaluated works, we utilize the default settings provided in their respective codes or papers and display the direct output results.

9.3 What Impact Does Peephole Optimization Have on Existing Binary Diffing Techniques?

In this section, we will conduct experiments and analyze the impact of peephole optimizations on binary differences using these two tools.

9.3.1 BinDiff

BinDiff, an extensively cited and widely acknowledged binary difference analysis tool [60, 61], utilizes disassembly code generated by IDA [32] as its input source and incorporates a three-tiered statistical system encompassing function, basic block, and control flow/call graph topological order. This approach enhances the efficiency of the matching process. The similarity score provided by BinDiff for comparing two binary codes ranges from 0.0 to 1.0, where higher scores signify greater similarity. This score constitutes a weighted sum of five metrics[1]: 35% for matched control flow graph

[1]BinDiff Manual: `https://www.zynamics.com/bindiff/manual/`.

edges, 25% for matched basic blocks, 20% for differences in call graph topological order, 10% for matched functions, and 10% for matched instructions.

BinDiff employs a general matching strategy that relies on a list of function attributes suitable for generating matches. This process begins at the global level by considering all functions of the binary and calculating the first attribute for each function. Following the initial global matching step, BinDiff examines parents (callers) and children (callees) of each new match, executing a ”drill down” step to match functions within the set of parents and children using the next best attribute. This procedure is repeated until all unmatched functions have been evaluated.

Function matching algorithms are organized according to the resulting match quality and can be classified into two categories: canonical per function and per edge. Edge matching seeks to match edges, representing calls in the call graph or jumps in the flow graph if the source and target function attributes correspond. This method produces superior matches, but it may be slower in instances where there are numerous edges per vertex in a graph.

Basic block matching, at the flow graph level, shares algorithmic similarities with function matching, where global attribute matching is succeeded by drill downs and matching in the reduced set of parents/children of matched basic blocks. Basic block matching algorithms are also arranged roughly by the resulting match quality.

In summary, BinDiff's matching strategy aims to deliver high-quality matches by employing various algorithms and considering different aspects of functions and basic blocks. It is crucial to recognize that peephole optimizations can influence binary code by modifying instructions within basic blocks or combining multiple basic blocks. These effects manifest in various code representations, such as basic block comparisons, control flow, and instruction sequence, consequently influencing the comparison outcomes.

Table 9.2: Experimental Results on Rotating Left Operation Program: Unoptimized Binary vs. Peephole Optimized Binary (by BinDiff)

	Similarity	Modification	Difference
Basic blocks	100%	/	/
Instructions	36.4%	6.8%	56.8%
Functions	100%	/	/
Calls	50%	/	50%
Jumps	/	/	/
Ultimate outcome	58%	/	42%

9.3.2 Analysis Results of Two Types of Peephole Dataset Employing BinDiff

In order to comprehensively investigate the impact of peephole optimization on two distinct types of binary code, we initially conducted separate experiments on the previously mentioned motivation examples, namely the rotate left operation algorithm and the population count algorithm. The experimental results are presented in the following tables (Table 9.2 and Table 9.3).

Example 1 pertains to the modification of instructions within a basic block. Following peephole optimization, the ROL instruction supersedes the original instruction sequence. As a result, BinDiff's experimental findings indicate that a mere 36.4% of the instruction sequence remains similar post-optimization, with 6.8% modified and 56.8% entirely altered. Calls are identified as 50% modified, potentially due to the rearrangement of instruction sequences. In contrast, the basic block and function are recognized as 100% similar. Moreover, no differences in Jumps are observed after optimization. Consequently, Example 1 exemplifies a peephole optimization variant that impacts the instruction sequence within a basic block, causing substantial syntactic

Table 9.3: Experimental Results on Population Count Program: Unoptimized Binary vs. Peephole Optimized Binary (by BinDiff)

	Similarity	Modification	Difference
Basic blocks	50%	/	50%
Instructions	26.2%	11.9%	61.9%
Functions	100%	/	/
Calls	50%	/	50%
Jumps	/	/	100%
Ultimate outcome	48%	/	52%

discrepancies between the optimized and preceding instruction sequences. Despite appearing dissimilar, they represent the same program, with one binary having undergone peephole optimization. These incongruent disparities contribute to a decline in binary similarity results.

Example 2 encompasses modifications spanning multiple basic blocks. In the aftermath of peephole optimization, the POPCNT instruction supersedes the original sequence of instructions, effectively consolidating several basic blocks into a single unit. This sharply diverges from Example 1. BinDiff's experimental findings disclose that only 50% of the basic blocks are identified as similar, while the remaining 50% are entirely distinct. The similarity of the instruction sequences further decreases, as merely 26.2% of instructions are identified as similar, 11.9% of instructions are modified, and 61.9% of instructions are completely different. Jumps are recognized as 100% different, signifying a total transformation in control flow. The call is identified as a 50% modification, potentially attributed to a rearrangement of the instruction sequence. In contrast, the functionality is considered 100% similar. Thus, Example 2 exemplifies a peephole optimization variant that influences the instruction sequence across multiple basic blocks, yielding significant differences between the optimized and preceding basic blocks. The ramifications on control flow and basic blocks contribute to a further reduction in binary similarity results. Ultimately, the final outcome reveals that the similarity between the binary files pre- and post-optimization is a mere 48%.

Examples 1 and 2 illustrate the effects of peephole optimization on binary code, revealing substantial differences between the optimized and preceding instruction sequences. Example 1 highlights the impact on instructions within a basic block, while Example 2 demonstrates modifications that span multiple basic blocks. Both scenarios result in a decline in binary similarity results, with Example 2 showing even more significant differences due to the ramifications on control flow and basic blocks. These findings prompt further investigation into the collective effects of peephole optimization. To this end, we will conduct experiments using hundreds of peephole optimization program units to observe the influence of peephole optimization on binary code and the potential chain reactions that emerge when multiple optimizations are implemented simultaneously.

In the subsequent section, we will carry out experiments employing two different types of datasets. The first type, denoted as type 1 datasets, encompasses program units in which the instruction sequence inside basic blocks undergoes modifications after the application of peephole optimization. Following this, we will scrutinize the second dataset type, comprising program units that, when subjected to peephole optimization, exhibit alterations in multiple basic blocks and control flow. Through the utilization of these datasets, our objective is to comprehensively examine the impact of peephole optimization on diverse program structures and to augment our comprehension of its ramifications on binary diffing. The experimental results are presented in the following tables (Table 9.4 and Table 9.5).

The type 1 dataset encompasses instances in which peephole optimization alters the instructions within basic blocks. Following the application of peephole opti-

Table 9.4: Experimental Results on Type 1 Datasets: Unoptimized Binary vs. Peephole Optimized Binary (by BinDiff)

	Similarity	Modification	Difference
Basic blocks	50%	10%	40%
Instructions	1.9%	19.4%	78.6%
Functions	63.6%	/	36.4%
Calls	0.3%	49.6%	50.1%
Jumps	33.3%	16.7%	50%
Ultimate outcome	59%	/	41%

mization, the original instruction sequences are supplanted by more streamlined and efficient instructions. Experimental outcomes utilizing BinDiff indicate that post-peephole optimization, 50% of the basic blocks are identified as similar to their unoptimized counterparts, 10% are recognized as modified, and 40% are classified as entirely different.

Intriguingly, the instruction sequences after peephole optimization display a mere 1.9% similarity, with 19.4% identified as modified and an impressive 78.6% recognized as entirely distinct. Moreover, as numerous basic blocks experience modifications, the identification of the final functions is also impacted, with 63.6% acknowledged as similar and 36.4% identified as different. Calls are also substantially affected, with a scant 0.3% identified as similar and a remarkable 50.1% recognized as completely dissimilar, illustrating the extensive range of adjustments made to the instruction sequences.

The ultimate findings exhibit a 59% similarity and a 41% difference. These experimental results demonstrate that as a considerable number of instruction sequences are replaced following peephole optimization, various aspects, including basic block comparisons, instruction comparisons, and calls, are both directly and indirectly influenced. Ultimately, these effects are manifested in the final results.

Table 9.5: Experimental Results on Type 2 Datasets: Unoptimized Binary vs. Peephole Optimized Binary (by BinDiff)

	Similarity	Modification	Difference
Basic blocks	11.4%	34.3%	54.3%
Instructions	5.2%	36.5%	58.3%
Functions	70.6%	17.6%	11.8%
Calls	0.6%	64.2%	35.2%
Jumps	2.7%	45.4%	51.9%
Ultimate outcome	26%	/	74%

The type 2 program unit collection uses peephole optimization to modify multiple basic blocks and applies hardware-based streamlined instructions, resulting in the merging of several basic blocks. This differs significantly from Example 1. According to experimental results from BinDiff, only 11.4% of basic blocks were found to be similar, 34.3% were modified, and 54.3% were entirely different basic blocks. This indicates that peephole optimization has a significant impact on basic blocks. In terms of instruction sequences, only 5.2% of instructions were identified as similar, 36.5% were modified, and 58.3% were entirely different. As basic blocks are modified, functions are 70.6% similar, while jumps are 51.9% different, indicating a complete transformation of control flow. The calls were found to be 64.2% modified, possibly due to the rearrangement of the instruction sequences. Overall, the Type 2 program unit experiment demonstrates the significant impact of peephole optimization on basic blocks and instruction sequences. Only 11.4% of basic blocks were found to be similar, and there was a reduction in binary similarity results due to the impact on control flow. Therefore, the similarity between binary files before and after optimization was only 26%.

To summarize, the impact of peephole optimization on basic blocks and instruction sequences has been investigated in two different datasets. The Type 1 dataset involved the alteration of instructions within basic blocks, resulting in 50% similarity to the unoptimized counterparts, 10% modification, and 40% entirely differences. The instruction sequences after optimization showed a mere 1.9% similarity, with 19.4% identified as modified and 78.6% entirely distinct. The Type 2 program unit collection merged multiple basic blocks using hardware-based streamlined instructions, with only 11.4% of basic blocks being similar, while 34.3% were modified, and 54.3% were entirely different. The transformation of control flow resulting from peephole optimization led to a significant impact on binary similarity results. Next, we further investigate the impact of peephole optimization on basic blocks and instruction sequences based on a state-of-the-art AI-based binary tool.

9.3.3 DeepBinDiff

DeepBinDiff [64] is a binary code similarity analysis tool, that excels in identifying minute discrepancies at the instruction level within binary code. By employing deep learning methodologies, this tool conducts precise and efficient comparisons of binary code, outperforming conventional binary diffing tools in discerning alterations in control flow structures, obfuscation methods, and code enhancements.

The foundation of DeepBinDiff lies in an unsupervised deep neural network-based approach for binary diffing, which initially learns basic block embeddings through unsupervised deep learning. These embeddings encapsulate specific blocks with semantic and contextual information derived from the interprocedural control flow graph (ICFG). The prototype implementation demonstrates that these embeddings facilitate the efficient and accurate computation of basic block similarities, thereby ensuring logical coherence and readability in the context.

The process of generating basic block embeddings based on ICFGs (Interprocedural Control Flow Graphs) and feature vectors involves merging two ICFGs into one graph and modeling the problem as a network representation learning problem. The Text-associated DeepWalk (TADW) algorithm is employed to generate these embeddings. TADW is an unsupervised graph embedding learning technique that incorporates the features of vertices into the network representation learning process. It is an improvement over the DeepWalk algorithm, which only considers contextual information from a graph but not the node features.

Graph merging is proposed to address the limitations of running TADW separately for two graphs. Merging the two ICFGs into one graph allows for a more efficient matrix factorization and improves similarity detection. DeepBinDiff extracts string references and detects external library calls and system calls, creating virtual nodes for strings and library functions and drawing edges from the call sites to these virtual nodes. Basic block embeddings are generated using the TADW algorithm on the merged graph. DeepBinDiff feeds the merged graph and the basic block feature vectors into TADW for multiple iterations of optimization. The resulting basic block embeddings contain semantic information about the basic block itself as well as information from the ICFG structure. A k-hop greedy matching algorithm is introduced to address the limitations of linear assignment and benefit from the ICFG contextual information. This algorithm finds matching basic blocks based on the similarity calculated from basic block embeddings within the k-hop neighbors of already matched ones.

To sum up, the technical merits of DeepBinDiff have piqued our interest. By employing the TADW algorithm, graph merging, and k-hop greedy matching during the embedding generation process, DeepBinDiff creates basic block embeddings and

Table 9.6: Experimental Results on Two Examples: Unoptimized Binary vs. Peephole Optimized Binary (by DeepBinDiff)

Program	Total Nodes of Two Binaries	Matched Pairs	Matched Ratio
Left rotation operation	22	0	0%
Population count	26	0	0%

locates matching basic blocks within the merged ICFG. As a result, it has established itself as a state-of-the-art tool for detecting differences at the basic block level.

9.3.4 Analysis Results of Two Types of Peephole Dataset Employing DeepBinDiff

In order to maintain logical consistency and thoroughly examine the effects of peephole optimization on two distinct types of binary code, we initially conducted separate experiments on the aforementioned motivational examples, specifically focusing on the left rotation operation algorithm and the population count algorithm. It is important to note that, in order to preserve the integrity and consistency of the tool, we avoided making unauthorized modifications to the experimental configurations. The experiment was performed using default settings to ensure the accuracy of our results. The experimental findings are presented in Table 9.6.

Unexpectedly, the overall experimental results were not as anticipated, since DeepBinDiff was unable to identify any matches between the two example programs, which comprised 22 and 26 nodes in the motivational examples. Ultimately, none of these nodes matched. Moreover, to ensure the tool's usability, we analyzed the open-source binary files provided by the author, comparing the binary files[2] generated by Coreutils-5.93/chroot when compiled with different settings at optimization levels O0

[2]Official Repository for DeepBinDiff: `https://github.com/yueduan/DeepBinDiff/tree/master/experiment_data/coreutils/binaries/`.

Table 9.7: Experimental Results on Two Types of Datasets: Unoptimized Binary vs. Peephole Optimized Binary (by DeepBinDiff)

Program	Total Nodes	Matched Pairs	Matched Ratio
Type 1: Effect within a basic block	437	0	0%
Type 2: Effect across multiple basic blocks	424	9	4.2%

and O1. These two binaries contained a total of 1,180 nodes, and the final matching pairs amounted to 100, resulting in a matching rate of 16.9%. Upon analysis, we hypothesize that the low recognition rate may be attributed to a relatively low range value set for k-hop.

In the next step, we will conduct an analysis of two datasets, and the corresponding results can be found in Table 9.7. In the analysis of binary files before and after peephole optimization, DeepBinDiff failed to detect any matches among the 437 nodes in dataset type 1. In dataset type 2, DeepBinDiff identified 9 matching pairs out of 424 nodes. We have contacted the author of DeepBinDiff about the experimental results to confirm that this tool's results are correct, and we will update this part of the experimental results in the future if necessary. We are willing to contribute to the advancement of the binary analysis field through joint efforts. Moreover, we located the basic block embeddings based on the index values in the matching pair results. We observed and confirmed that significant differences exist in these basic block embeddings. These significant differences further confirm a low match rate of results.

Peephole optimization introduces additional complexity to analysis techniques based on basic block embeddings. Essentially, since peephole optimization not only thoroughly replaces the instruction sequence within an original basic block but can also merge multiple basic blocks, this can lead to two issues. First, due to the almost complete replacement, the two basic blocks before and after optimization do not match at all. Second, as a result of merging multiple basic blocks, the control flow changes and corresponding basic block pairs cannot be found. For example, the complete change in basic blocks may cause the two embedding vectors to differ significantly, leading to poor matching results. At the same time, the change of control flow and mismatch in the number of basic blocks will require more computational resources to handle the "m-to-n" embedding and matching processes.

9.4 Answer to RQ3

In summary, peephole optimization significantly influences existing binary difference techniques at both basic block and control flow levels, with performance degradation emphasizing the necessity of considering peephole optimization. We have chosen two of the most representative and renowned tools for our investigation: BinDiff, an industry and academic standard for over a decade, and DeepBinDiff, a cutting-edge unsupervised learning approach that exemplifies AI-based methods concentrating on basic block-level detection. This influence ultimately manifests itself in binary analysis tools. BinDiff results indicate that peephole optimization can cause 50% of optimized instructions within basic blocks to be unmatchable and lead to modifications spanning multiple basic blocks. Jumps and Calls identify over 50% of the differences. Simultaneously, peephole optimization introduces significant complexity to DeepBinDiff's binary analysis technique, resulting in a mismatch in the number of basic blocks and the replacement of entirely different new basic blocks, as well as control flow changes that affect matching strategies based on control flow levels. These mismatches influence basic block embedding vectors and matching algorithms. In conclusion, the modifications to basic blocks by peephole optimization impact the accuracy of binary comparison results, leading to unsatisfactory matching outcomes.

CHAPTER 10

Discussion and Threats to Validity

In our study, we employed Alive2, a tool specifically designed for capturing LLVM Intermediate Representation (IR), to investigate peephole optimization. Consequently, the efficacy and precision of our analysis are heavily dependent on the performance and accuracy of Alive2. It is important to note that our research primarily focuses on LLVM rather than the GNU Compiler Collection (GCC). This is because GCC lacks a mechanism comparable to LLVM's plugin functionality, which enables the capture of intermediate IR snapshots before and after optimization. This capability is crucial for conducting an in-depth analysis of the alterations in basic blocks and instructions during each optimization phase. Without such a feature, a comprehensive evaluation of peephole optimization in GCC becomes challenging.

Our investigation is predominantly centered on the x86-64 platform, rather than examining multiple architectures. The rationale behind this decision is the platform-dependent nature of peephole optimization. Each architecture has its unique instruction set and features, which directly influence the types of peephole optimization that can be applied. By focusing on a single platform, we can delve deeper into the specific optimizations related to that platform, providing a more detailed and concentrated analysis.

Finally, in our examination of peephole optimizations using state-of-the-art binary comparison tools, we selected two highly representative and well-regarded tools: BinDiff, an industry and academic standard for over a decade, and DeepBinDiff, an unsupervised learning approach that exemplifies AI-based methods concentrating on basic block-level detection. Our analysis does not encompass all existing tools, as some are similar and belong to the same category, while others are not open-source.

CHAPTER 11

Conclusion and Future Work

In this study, we employ a customized compiler-based translation verification tool to systematically analyze the implications of peephole optimizations. We find that the impact of this long-standing and widely used optimization technique on intraprocedural binary code differences has been underestimated. Our study underscores the importance of addressing the effect of peephole optimization on binary diffing analysis, as well as the limitations of current basic-block-centric comparison approaches. We encourage the research community to reconsider experimental designs that include peephole-optimized scenarios, enabling a more comprehensive evaluation of capabilities beyond peephole optimization. In the future, we will further explore the effects of other compiler optimizations on binary differences more extensively.

REFERENCES

[1] LLVM, "Pass," https://llvm.org/docs/WritingAnLLVMPass.html, [online].

[2] The LLVM Compiler Infrastructure, "LLVM's Analysis and Transform Passes," https://llvm.org/docs/Passes.html, [online].

[3] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably Correct Peephole Optimizations with Alive," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15), 2015.

[4] D. Menendez and S. Nagarakatte, "Termination-checking for LLVM Peephole Optimizations," in Proceedings of the 38th International Conference on Software Engineering, 2016.

[5] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "A Study on Malicious Software Behaviour Analysis and Detection Techniques: Taxonomy, Current Trends and Challenges," Future Generation Computer Systems, vol. 130, pp. 1–18, 2022.

[6] P. P. Ray, "Internet of Things for Smart Agriculture: Technologies, Practices and Future Direction," Journal of Ambient Intelligence and Smart Environments, vol. 9, no. 4, pp. 395–420, 2017.

[7] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 32, no. 6, Aug. 2010.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of)

The Art of War: Offensive Techniques in Binary Analysis," in Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16), 2016.

[9] X. Meng and B. P. Miller, "Binary Code is Not Easy," in Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16), 2016.

[10] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-platform Binary," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), 2018.

[11] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18), 2018.

[12] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary Code Clone Detection Across Architectures and Compiling Configurations," in Proceedings of the 25th International Conference on Program Comprehension (ICPC'17), 2017.

[13] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and T. H. B. Kuan, "BinGo: Cross-Architecture Cross-OS Binary Search," in Proceedings of the 2016 ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16), 2016.

[14] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS'16), 2016.

[15] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging Semantic Signatures for Bug Search in Binary Programs," in Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14), 2014.

[16] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15), 2015.

[17] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P'08), 2008.

[18] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills," in Proceedings of the 39th International Conference on Software Engineering (ICSE'17), 2017.

[19] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "PatchScope: Memory Object Centric Patch Diffing," in Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20), 2020.

[20] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[21] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences," IEEE Transactions on Software Engineering, vol. 41, no. 12, 2015.

[22] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), 2014.

[23] D. K. Chae, J. Ha, S. W. Kim, B. Kang, and E. G. Im, "Software Plagiarism Detection: A Graph-based Approach," in Proceedings of the 22nd ACM

International Conference on Information & Knowledge Management (CIKM'13), 2013.

[24] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Behavior Based Software Theft Detection," in Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), 2009.

[25] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17), 2017.

[26] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate Comparison of Binary Executables," in Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13), 2013.

[27] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis," in Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11), 2011.

[28] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying Dormant Functionality in Malware Programs," in Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10), 2010.

[29] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors," in Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10), 2010.

[30] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale Malware Indexing Using Function-call Graphs," in Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), 2009.

[31] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09), 2009.

[32] Hex-Rays, "IDA Pro Dissasember," https://www.hex-rays.com/products/ida, [online].

[33] I. U. Haq and J. Caballero, "A Survey of Binary Code Similarity," ACM Computing Surveys (CSUR), vol. 54, no. 3, pp. 1–38, 2021.

[34] R. Zak, E. Raff, and C. Nicholas, "What Can N-grams Learn for Malware Detection?" in 2017 12th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, 2017, pp. 109–118.

[35] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned," IEEE Transactions on Software Engineering, vol. 49, no. 4, 2022.

[36] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study," in Proceedings of the 42th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21), 2021.

[37] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 4, 2022.

[38] P. Knijnenburg, T. Kisuki, and M.F.P.O'Boyle, "Iterative Compilation," in Proceedings of the 2002 International Workshop on Embedded Computer Systems, 2002.

[39] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century," The Journal of Supercomputing, vol. 23, no. 1, 2002.

[40] W. M. McKeeman, "Peephole Optimization," Communications of the ACM, vol. 8, no. 7, pp. 443–444, 1965.

[41] S. Bansal and A. Aiken, "Automatic Generation of Peephole Superoptimizers," in Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), 2006.

[42] P. Chakraborty, "Fifty Years of Peephole Optimization," Current Science, 2015.

[43] S. Bansal and A. Aiken, "Binary Translation Using Peephole Superoptimizers," in Proceedings of the 8th USENIX on Operating Systems Design and Implementation (OSDI '08), 2008.

[44] L. T. Keith Cooper, Engineering a Compiler, 3rd ed. Morgan Kaufmann, 2022.

[45] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr, "Alive2: Bounded Translation Validation for LLVM," in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21), 2021.

[46] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations," Algorithmica, vol. 1, no. 1-4, pp. 251–266, 1986.

[47] Y. David, N. Partush, and E. Yahav, "Statistical Similarity of Binaries," in Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16), 2016.

[48] D. Xu, J. Ming, and D. Wu, "Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping," in Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17), 2017.

[49] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven Equivalence Checking," in Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'13), 2013.

[50] S. Wang and D. Wu, "In-memory Fuzzing for Binary Code Similarity Analysis," in Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17), 2017.

[51] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14), 2014.

[52] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs," in Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12), 2012.

[53] Y. David, N. Partush, and E. Yahav, "Similarity of Binaries through Re-Optimization," in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17), 2017.

[54] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-based Bug Search for Firmware Images," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16), 2016.

[55] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs," in Proceedings of the 26th Network and Distributed System Security Symposium (NDSS'19), 2019.

[56] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detec-

tion," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17), 2017.

[57] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αDiff: Cross-version Binary Code Similarity Detection with DNN," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), 2018.

[58] S. Bansal and A. Aiken, "Automatic Generation of Peephole Superoptimizers," ACM SIGARCH Computer Architecture News, vol. 34, no. 5, pp. 394–403, 2006.

[59] D. Menendez, S. Nagarakatte, and A. Gupta, "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM," in Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23. Springer, 2016, pp. 317–337.

[60] H. Flake, "Structural Comparison of Executable Objects," in Proceedings of the 2004 GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'04), 2004.

[61] Google LLC, "BinDiff: Graph Comparison for Binary Files," https://www.zynamics.com/bindiff.html, 2017.

[62] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically Finding Semantic Differences in Binary Programs," in Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08), 2008.

[63] Y. David and E. Yahav, "Tracelet-based Code Search in Executables," in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), 2014.

[64] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," in Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20), 2020.