ABSTRACT

Title of Dissertation: Robust Low-Overhead Binary Rewriting: Design, Extensibility, And Customizability

Amir Majlesi Kupaei
Doctor of Philosophy, 2021

Dissertation Directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering

Binary rewriting is the foundation of a wide range of binary analysis tools and techniques, including securing untrusted code, enforcing control-flow integrity, dynamic optimization, profiling, race detection, and taint tracking to prevent data leaks. There are two equally important and necessary criteria that a binary rewriter must have: it must be robust and incur low overhead. First, a binary rewriter must work for different binaries, including those produced by commercial compilers from a wide variety of languages, and possibly modified by obfuscation tools. Second, the binary rewriter must be low overhead. Although the off-line use of programs, such as testing and profiling, can tolerate large overheads, the use of binary rewriters in deployed programs must not introduce significant overheads; typically, it should not be more than a few percent. Existing binary rewriters have their challenges: static rewriters do not reliably work for stripped binaries (i.e., those without relocation information), and dynamic rewriters suffer from high base overhead. Because of this

high overhead, existing dynamic rewriters are limited to off-line testing and cannot be practically used in deployment.

In the first part, we have designed and implemented a dynamic binary rewriter called RL-Bin, a robust binary rewriter that can instrument binaries reliably with very low overhead. Unlike existing static rewriters, RL-Bin works for all benign binaries, including stripped binaries that do not contain relocation information. In addition, RL-Bin does not suffer from high overhead because its design is not based on the code-cache, which is the primary mechanism for other dynamic rewriters such as Pin, DynamoRIO, and Dyninst. RL-Bin's design and optimization methods have empowered RL-Bin to rewrite binaries with very low overhead (1.04x on average for SPECrate 2017) and very low memory overhead (1.69x for SPECrate 2017). In comparison, existing dynamic rewriters have a high runtime overhead (1.16x for DynamoRIO, 1.29x for Pin, and 1.20x for Dyninst) and have a bigger memory footprint (2.5x for DynamoRIO, 2.73x for Pin, and 2.3x for Dyninst). RL-Bin differentiates itself from other rewriters by having negligible overhead, which is proportional to the added instrumentation. This low overhead is achieved by utilizing an in-place design and applying multiple novel optimization methods. As a result, lightweight instrumentation can be added to applications deployed in live systems for monitoring and analysis purposes.

In the second part, we present RL-Bin++, an improved version of RL-Bin, that handles various problematic real-world features commonly found in obfuscated binaries. We demonstrate the effectiveness of RL-Bin++ for the SPECrate 2017 benchmark obfuscated with UPX, PECompact, and ASProtect obfuscation tools.

RL-Bin++ can efficiently instrument heavily obfuscated binaries (overhead averaging 2.76x, compared to 4.11x, 4.72x, and 5.31x overhead respectively caused by DynamoRIO, Dyninst, and Pin). However, the major accomplishment is that we achieved this while maintaining the low overhead of RL-Bin for unobfuscated binaries (only 1.04x). The extra level of robustness is achieved by employing dynamic deobfuscation techniques and using a novel hybrid in-place and code-cache design.

Finally, to show the efficacy of RL-Bin in the development of sophisticated and efficient analysis tools, we have designed, implemented, and tested two novel applications of RL-Bin; An application-level file access permission system and a security tool for enforcing secure execution of applications. Using RL-Bin's system call instrumentation capability, we developed a fine-grained file access permission system that enables the user to define separate file access policies for each application. The overhead is very low, only 6%, making this tool practical to be used in live systems. Secondly, we designed a security enforcement tool that instruments indirect control transfer instructions to ensure that the program execution follows the predetermined anticipated path. Hence, it would protect the application from being hijacked. Our implementation showed effectiveness in detecting exploits in real-world programs while being practical with a low overhead of only 9%.

Robust Low-Overhead Binary Rewriting: Design, Extensibility, and Customizability

by

Amir Majlesi Kupaei

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Manoj Franklin
Professor Bruce Jacob
Professor Donald Yeung
Professor Adam Porter, Dean's Representative

Acknowledgments

First of all, I am grateful beyond words to my family, who encouraged and inspired me, not just along this Ph.D. journey but always. I am thankful to my parents, who have always supported me throughout my life. I would not be going to graduate school, nor doing this research, were it not be for their support and encouragement.

Next, I want to thank my wonderful wife, Zahra, for bringing happiness into my life and supporting me through the endless years of graduate school. I am truly grateful for your love, support, and patience. Without you, I would not be able to thrive in my doctoral program.

I want to thank my incredible sister, Alaleh, who has always been on my side and supported me throughout my life. I appreciate her kind support, which she gives me remotely through Skype, even when we are far apart.

Next, I am thankful to my best friend, Ali, for his precious support during my graduate and undergraduate studies. This would have never been possible without his help and encouragement. I would also like to extend my thanks to my friends Hamid Reza, Kamyar, Amir Hossein for all their help and support. I appreciate their friendship.

Next, I would like to thank my friend Danny Kim, whose presence in the lab

and his sound ideas were crucial for me during the first four years. I hope that I have been as helpful to him as he was to me.

I want to thank the fellow students Kapil Anand, Khaled Elwazeer, and Aparna Kotha, who were in this research group before me. They initially helped me with the Secondwrite binary rewriter and gave me valuable insight into the implementation of RL-Bin.

In the end, I want to thank my advisor, Professor Rajeev Barua, for guiding, encouraging, and advising me during all these years. I am appreciative of his commitment to his students. My thesis is the fruit of our lengthy discussions and meetings and his novel and creative suggestions and ideas.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
APM	Application Performance Monitoring
ARM	Advanced RISC Machine
BB	Basic Block
BIRD	Binary Interpretation using Runtime Disassembly
BSS	Block Starting Symbol
CFG	Control Flow Graph
CFI	Control Flow Integrity
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CTI	Control Transfer Instruction
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DDR	Double Data Rate
DLL	Dynamically Loaded Library
DRM	Digital Rights Management
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU is Not Unix
GOT	Global Offset Table
HTML	Hypertext Markup Language
HW	Hardware
IAT	Import Address Table
ICC	Intel C++ Compiler
IDA	Interactive Disassembler
IO	Input Output

IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just In Time
KLOC	Kilo Lines Of Code
MSIL	Microsoft Intermediate Language
OS	Operating System
PDB	Program Data Base
PDF	Portable Document Format
PE	Portable Executable
PIC	Position Independent Code
PLT	Program Linkage Table
PNSD	Provably Not Stack Derived
RISC	Reduced Instruction Set Computer
ROP	Return Oriented Programming
SEH	Structured Exception Handling
SPEC	Standard Performance Evaluation Corporation
SSDT	System Service Dispatch Table
SW	Software
TLB	Translation Lookaside Buffer
UPX	Ultimate Packer for eXecutables
VBA	Visual Basic for Applications
VEH	Vectored Exception Handling
XML	eXtensible Markup Language

Chapter 1: Introduction

1.1 Motivation

There are several reasons why it is desirable to instrument or modify code that is directly executed in deployment, ranging from application performance monitoring (APM) [1], resource monitoring [2], security policy enforcement [3, 4, 5, 6, 7], vulnerability patching [8], dynamic information flow tracking [9], and performance optimization [10, 11, 12, 13]. Taken together, these form important sectors of the software industry: for example, the application performance monitoring market alone is a $3.5B/year market, and security policy enforcement on low-level is also a multi-billion dollar market.

There are two types of code that is directly executed in deployment: 1. Interpreted code and 2. Binary code. Interpreted code is the code that executes only in a software run-time interpreter (also known as an execution engine.) Examples include Java bytecode, Python code, C# bytecode, and Javascript code. Instrumenting or modifying such code is straightforward: most language interpreters provide methods to trigger user-specified actions when certain types of code instructions or library calls are executed. Using this capability, entire industries have arisen for instrumenting interpreted languages, such as in application performance monitoring

and security policy enforcement. This instrumentation and modification capability, along with language-level advantages such as portability and modularity, are the reasons why interpreted languages have become so popular in practice.

However, binary code (also known as machine code) has remained stubbornly widespread. Binary code is the code that executes directly on the hardware using machine code instructions. Binary code can theoretically be produced from any language, but is typically produced not only from older languages like C, C++, Fortran, and Cobol; but is also often produced from popular modern languages such as Go, Erlang, VisualBasic, Swift, and Objective C.

Binary code is especially predominant in two types of code: IP-protected code, and high-performance code. First, IP-protected code is code that is sold by companies to outside parties. Companies in nearly all cases want to protect their intellectual property, so they do not want to reveal their source code to their customers or third parties. Unfortunately this rules out most interpreted codes – many interpreted codes used in deployment either *are* source code, such as Python or Javascript; or source code can be easily recovered from them, such as from Java and C++bytecodes. As a result, interpreted languages are very common for internally used code at companies, as well as cloud-based code, but is nearly absent in distributed commercial codes. For example, nearly all programs that come pre-packaged on a new laptop, as well as most commercial programs that customers download and buy, are binary code. *Such distributed, IP-protected code is the most widespread use of binary code today.* Second, high-performance programs such as those in the domains of image processing, financial transactions, machine learning, and scientific simulation codes

are often deployed in binary to ensure the highest execution speed.

It is important to gain the same benefits of instrumentation and modification mentioned above for binary code that interpreted languages have long enjoyed. To do so, we need a tool that can instrument and modify binary code, namely, a binary rewriter.

In general, the reason for the great interest in research in binary rewriting is that it offers many additional advantages over compiler-produced optimized binaries:

- Inter-procedural Optimization Capability. Although compilers, in theory, can do whole-program optimizations, the reality is that they often do not. Many commercial compilers – even highly optimizing ones – limit themselves to the separate compilation, where each function (and sometimes each file) is compiled in isolation. For example, GCC, the most widely used open-source compiler used commercially, compiles each function in isolation even with the highest optimization level. Research papers have presented whole-program optimizations, but in limited contexts, with much remaining to be done. Binary rewriters always have access to the complete application, including libraries. Recognizing the deficiency regarding whole-program optimization in commercial compilers and research, existing rewriters have proposed many whole-program optimizations. They have demonstrated promising results on even highly optimized binaries [14, 15] without excessive run-time. Having the ability to rewrite arbitrary binaries will give external innovators the flexibility to improve upon the offerings of commercial whole-program compilers

– something they cannot do today.

- Extended Economic Feasibility. It is cheaper to implement a code transformation once for an instruction set in a binary rewriter rather than repeatedly for each compiler for the instruction set. For example, the ARM instruction set has over thirty compilers available for it, and the x86 has a similarly large number of compilers from different vendors and different source languages. The high expense of repeated compiler implementation usually cannot be supported by a small fraction of the demand.

- Source Language and Compiler Compatibility. A binary rewriter works for code produced from any source language by any compiler.

- Security Enforcement Capability. A binary rewriter that can rewrite binaries without relocation information can insert security checks into the binary. This is not the case for a compiler since a malicious developer can avoid the security checks by merely using a non-trusted compiler without security checks.

- Hand-coded Assembly Support. Code transformations cannot be applied by a compiler to hand-coded assembly routines, since they are never compiled. In contrast, a binary rewriter can transform such routines.

In this dissertation, we have designed and developed RL-Bin, a dynamic binary rewriter which is capable of rewriting all benign binaries with very low overhead. RL-Bin will find use in implementing a variety of applications of binary rewriting. For example, researchers have proposed binary rewriting-based methods for securing

untrusted code [5, 16, 17], enforcing control flow integrity [18, 19], protocol reverse engineering [20, 21], implementing software transactional memory [22], binary randomization [23, 24, 25, 26], preventing control flow attacks [2, 27, 28, 29], automated vulnerability repair [8, 30], profiling and race detecting tools [31], memory tracing to identify cache inefficiencies [32], automatic program parallelization [33, 34, 35, 36, 37], and taint tracking to prevent sensitive data leaks [38, 39, 40]. Taken together, binary rewriting technologies offers great existing features, and an almost unlimited future of yet-undiscovered opportunities, bounded only by the creativity of researchers.

Figure 1.1 shows some of the use cases of RL-Bin. In Chapter 8 we have designed and implemented prototypes of some of these use cases.

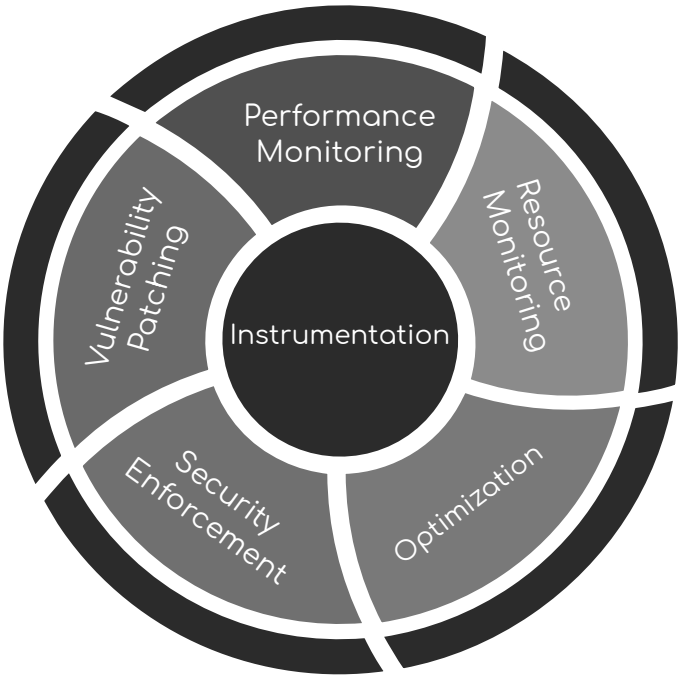


Figure 1.1: Applications of Binary Rewriters

1.2 Criteria and Trade-Offs in Building a Binary Rewriter

There are *two equally important and necessary criteria that a binary rewriter must have: it must be robust, and it must incur low overhead.* First, a binary rewriter must work for different type of binaries, including those produced by commercial compilers from a wide variety of languages, and possibly modified by obfuscation tools. Second, the binary rewriter must be low overhead. Although the *off-line use of programs*, such as in testing and profiling, can tolerate large overheads, the use of binary rewriters in *deployed programs* must not introduce significant overheads; typically, it should not be more than a few percent [41].

Unfortunately there is no existing method today to modify all benign binary code in a manner that is both robust and low overhead. To understand why, consider that there are two types of binary rewriters: static vs. dynamic. Static rewriting refers to approaches which take an executable binary program as input, and without running it, produce another (rewritten) binary program as output that has the same functionality as the input program, but is enhanced in some way, for example in improving its run-time, memory use, or security. Dynamic rewriters change the binary code during its execution, but never produce a binary program as output. Instead they modify the binary code in memory, either in-place, or in a copy of the code memory.

Static rewriters are not robust. Static rewriters can have very low overhead, but are not robust – they often do not work for certain types of benign programs, including those containing dynamically generated code, self-modifying

code, and obfuscated work. As our related work section details, 24% of commercial benign programs we measured had dynamically generated code, and 1% had obfuscated code. Several schemes do not even work for simpler programs with indirect branches whose targets cannot be statically determined, especially for stripped binaries (*i.e.* those without relocation information). Stripped binaries are predominant in commercial third-party binaries. What is worse is that for all existing rewriters, they cannot predict beforehand if they will work or not – they just simply stop working without warning. When that happens, the program may crash, or it may work, but without the enhancements in the modified binary code.

Dynamic rewriters have high overhead. In contrast, dynamic rewriters are robust, but have high overheads, usually ranging from 20% to several hundred percent. Dynamic rewriters are robust because they discover all code at run-time. However, they incur high overhead since most of them maintain a code cache, where rewritten copies of code blocks are stored and executed from. Maintaining a code cache results in high overheads not only from the copying of code, but from the requirement to translate code addresses that are targets of control transfer instructions, since those CTI targets now need to be changed at run-time because they are now executed from a copy of the code, instead of the original code.

As a result of the drawbacks, *there is no rewriter today that meets the critically important, non-negotiable requirements of robustness and low overhead execution for deployment use.* As a result, binary rewriters are generally not used in deployment today on third-party programs, since for those programs, usually no guarantees can be made on how they were compiled.

Dynamic rewriters, such as DynamoRIO[42], copy all the code that executes into another memory region called a *code cache*. The code cache is useful because it ensures robustness; the program still works If a piece of data is mistakenly assumed to be code and rewritten. The reason is that the code cache was changed and the original copy of the code segment is still unchanged.

The overhead of dynamic rewriters is caused by two factors. First, copying the code into the code cache is expensive at run-time. Second, and more seriously, the target addresses of an indirect Control Transfer Instruction (CTI) must be translated at run-time because the locations of code have changed to be in the code cache instead. Such indirect jumps or calls are actually very common – they mostly arise from return instructions, function pointer calls, and calls to virtual functions in object-oriented languages, such as C++. This translation process is inevitable for DynamoRIO since the original destination address in the program is different from the address of the rewritten code inside the code cache.

1.3 Robust, Low-Overhead Binary Instrumentation

Consequent to the needs above, we developed RL-Bin, our novel dynamic binary rewriter that ensures both robust behavior and low-overhead execution. *RL-Bin provides guarantees that all benign programs execute correctly and that all the code in the program is rewritten.* First, it ensures low-overhead execution because it rewrites code in-place in memory, thus avoiding a code cache and its overhead of address translation and copying of code. Second, it is robust, because it monitors

every control transfer, so no portion of the binary is rewritten until we know it is code because we execute a control transfer to it at run time. The overhead of monitoring every control transfer is reduced in two ways: (a) the monitoring code removes itself after the potential code it monitors is proven to be code the first time it is executed; and (b) our design optimizes away some monitoring code using dynamic JIT-based optimizations that do not rely on unsafe assumptions on static code.

Figure 1.2 shows the criteria for developing a binary rewriter. As illustrated existing dynamic rewriters are robust but have high overhead and existing static rewriters have low overhead but are not robust. RL-Bin is both robust and has low run-time overhead.

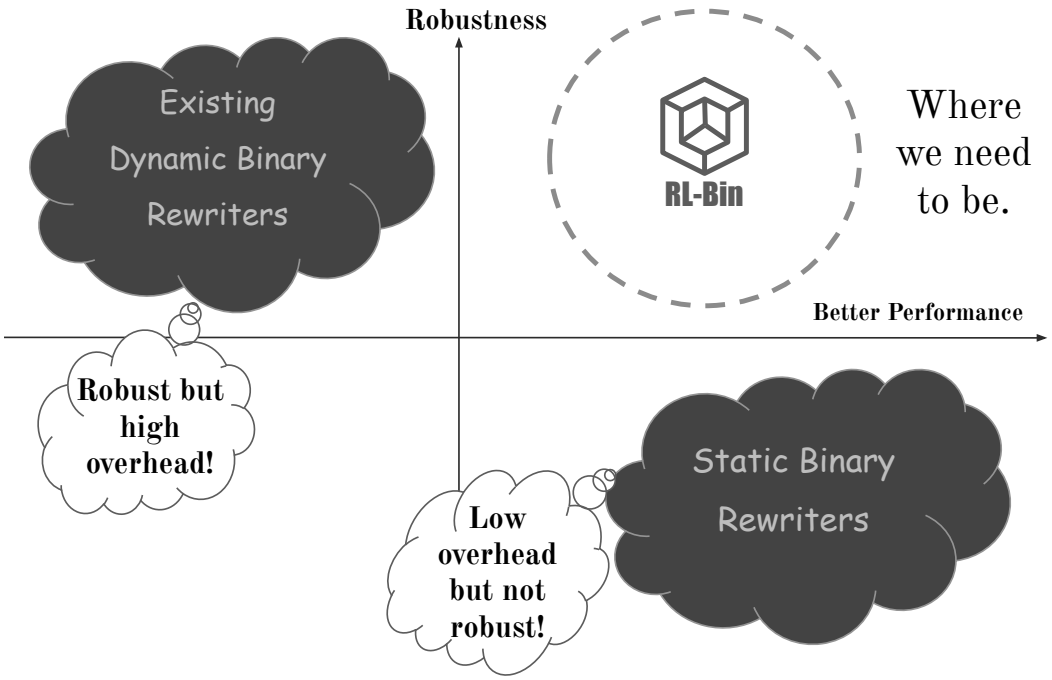


Figure 1.2: Criteria for Building a Binary Rewriter

RL-Bin supports several types of obfuscation, in addition to dynamically gen-

erated and self-modifying code. As a result, it is robust enough to be used for benign third-party applications. Also, we have designed and implemented several optimizations, so it has very low overhead. Throughout this thesis we present the following contributions.

- Chapter 3
  - Design and development of the first low overhead dynamic binary rewriter that can handle stripped binaries without relocation or debug information, containing self-modifying or dynamically-generated code or obfuscation.
  - An innovative method that tracks the execution of code dynamically by anticipating future control flow to the new code, and adding instrumentation and breakpoints to process such new code when discovered.
  - Using a novel dynamic method to eliminate the overhead of breakpoints, once the new code is discovered.
  - The above design is unlike other dynamic rewriters that translate indirect control transfer addresses to their copies in a code cache.
  - The result is the first In-Place dynamic binary rewriter – which does not use a code cache – that combines the robustness and coverage of a dynamic rewriter with the low overhead of a static rewriter.
- Chapter 4
  - Using Just-In-Time (JIT) dynamic analysis of the discovered code and

traditional data flow analysis concepts, to find "Safe" functions and further reduce the overhead by eliminating redundant checks.

  - In-depth analysis and exploration of trade-offs for several optimization methods to change the impact on memory consumption, including fine-tuning the parameters to get the best result.

- Chapter 5

  - An extensive study of problematic features in obfuscated binaries and the issues caused for binary rewriting tools.

- Chapter 6

  - Development of several innovative methods to enhance RL-Bin, empowering it to handle all kinds of benign binaries, including the ones that are heavily obfuscated.

  - Design and implementation of methods for dynamically deobfuscating binaries while they are being executed.

  - Design of a hybrid code-cache and in-place prototype that adapts itself based on the characteristics of a binary.

- Chapter 7

  - Extensive testing and run-time and memory overhead comparison with DynamoRIO, Pin, and Dyninst for SPEC CPU2017 benchmark with over 7 million lines of code in C, C++, and Fortran, compiled with Microsoft

Visual Studio, GCC, and ICC compilers. For obfuscated binary testing, benchmark applications are obfuscated with three of the most common obfuscation tools: UPX, PECompact, and ASProtect.

- Chapter 8
  - Full design and implementation of an application-level file access permission system as a use case of RL-Bin. This tool is built on system call instrumentation capability of RL-Bin and enables the user to define and enforce separate file access policies for different applications.
  - Design and development of a security tool for secure execution of applications as another use case of RL-Bin. This tool ensures that the original control flow and calling convention regarding the return instructions is properly enforced.

1.4 RL-Bin Advantages

RL-Bin will have the following advantages over existing binary rewriters:

- Does not require relocation information. Existing binary rewriters require relocation information, but as explained, most commercial binaries lack this information. As a result, only the original developers can rewrite those programs since only they have access to the object files, which need to be re-linked to produce binaries with relocation information. In contrast, RL-Bin can be applied by anyone to any binary executable.

- Can be applied to legacy applications. Existing binary rewriters cannot rewrite legacy binaries since virtually all binaries lack relocation information. Moreover, recompilation from source is often not possible since source code is often not readily available for legacy code. Our rewriter will rewrite legacy binaries without relocation information and source code.

- Can rewrite 100% of the binary code. Existing rewriters, even with relocation information, cannot rewrite 100% of a binary's code since they can only rewrite what they can prove is code. The difficulty is that data may be buried in the code section, which will break the program if rewritten. Hence rewriters must be conservative if they cannot prove that a portion of the binary is code and not rewrite it.

- Can be used to enforce security on untrusted code. Since existing static binary rewriters can only be used with developer cooperation, it is not feasible to enforce security properties on code from untrusted developers. This is because an untrusted developer may not provide relocation information, leaving us unable to rewrite the binary. However, any end-user can apply binary rewriting to enforce security on any code by using RL-Bin, including untrusted code. A malicious developer cannot avoid this. Moreover, since 100% of the binary's code can be rewritten, an attacker cannot hide malicious code in binaries by making it appear like it might be data to avoid rewriting.

- Can rewrite obfuscated binaries. Obfuscation is a technique used to mislead attempts to reverse-engineer the code, primarily by making it appear that

code is data, and vice-versa. Obfuscation is commonly used for high-level representations such as Java bytecode and Microsoft's MSIL, since they are close to the source. Recently obfuscation has become more prevalent in binary code. Existing binary rewriters cannot rewrite obfuscated binaries correctly. We have devised an innovative method that correctly rewrites the obfuscated code.

1.5 Outline of Thesis

The dissertation is structured as follows: Chapter 2 introduces some of the background knowledge needed for the subsequent chapters and also discusses the capabilities and limitations of RL-Bin. Chapter 3 outlines the base algorithm of RL-Bin and demonstrate our dynamic code discovery and execution routines. Chapter 4 describe the optimization methods designed to reduce the overhead of RL-Bin. We have also explored the trade-offs that exist in some of our optimization method. RL-Bin can be configured to decrease run-time or memory overhead. In Chapter 5, we have described obfuscation techniques and the issues they create for binary rewriting. Chapter 6 introduces RL-Bin++ which is an extension of RL-Bin that can handle obfuscated binary code with comparatively low overhead. Chapter 7 demonstrates the results of our evaluation of RL-Bin and RL-Bin ++ and compare them to DynamoRIO, Pin, and Dyninst. Chapter 8 looks into some use cases of RL-Bin including the following; application-level file access permission tool, secure execution by restricting RETs, collect run-time properties for end-point security

tool, generating guaranteed trusted disassembly, debugging and patching in deployment, and just-in-time analysis and optimization tool. In Chapter 9 we demonstrate the capabilites of RL-Bin's application programming interface and compare its runtime overhead to that of DynamoRIO. Chapter 10 describes the related works in detail. Finally, Chapter 11 looks ahead at potential future work and concludes the thesis.

Chapter 2: Background

In this chapter, we will go over structure of memory image of a binary application and then we review existing disassembly methods, since disassembly is an important step in binary rewriters. Then we introduce the basic concepts about some troublesome features that may be present in benign programs, since those will need to be handled by any robust binary rewriter. Finally, in the last section we briefly go over binaries with features for which RL-Bin might fail to instrument properly, hence we have introduced RL-Bin++ in Chapter 6.

2.1 Memory Image of a Binary Application

The memory image of a binary application consists of code segment, data segment, and some other memory areas that will be described in the following paragraphs.

The code segment, which is also known as a text segment, is where a portion of an executable file that contains instructions. It usually has read and execute permissions only when loaded to memory. One of the main challenges in binary rewriting is that it is possible to have data in the code segment. Every correct binary rewriter should only modify instructions and not the memory locations that contain

data. As a result, having data in the code segment means that instrumentation in an address can only be done after making sure that the memory location contains only code and no data.

The data segment contains global or static variables which have an initial value. These include any global variable that is not defined in a function, or static variables that are defined in a function but with static prefix so they retain their address across calls. One challenge for binary rewriters is that sometimes programs unpack the code in the data segment, and start executing code from there. Static binary rewriters would not be able to see this code, since it is generated only after the execution of the program begins.

Other memory areas in a running binary's image include the BSS segment, and heap and stack areas. The BSS segment is adjacent to the data segment and contains uninitialized data. This segment contains all variables that are not initialized or initialized to zero. The heap area contains dynamically allocated memory, and commonly begins after previous segments and grows to larger addresses. The heap area is shared between threads, shared libraries, and dynamically loaded modules. The stack area contains the program stack. The stack pointer keeps track of the top of the stack. The stack frame is the portion of the stack for a function, and contains its local variables, temporaries, return address, and outgoing arguments.

2.2 Existing Disassembly Techniques

Disassembly is a key step in static binary rewriting, so we discuss disassembly techniques, which sheds light on several of the difficulties with static binary rewriting.

2.2.1 Linear Sweep

Linear Sweep begins disassembly at the entry point into the code section of a binary. This entry point is provided by common executable file formats such as the Windows PE format. Each instruction is then decoded in sequence until the end of the section, or until an error occurs. An advantage of linear sweep disassembly is that it ensures complete code coverage, making it suitable for human viewing of disassembly output. Its downside is that it can mistake data for code, such as after an unconditional jump, leading to incorrect rewriting. Hence linear sweep is unacceptable by itself for rewriters.

2.2.2 Recursive Traversal

Recursive Traversal only disassembles an instruction when we find a control-flow path to that instruction. To do so, it starts disassembly at the binary code's entry point, but recognizes control transfers such as branches, jumps and calls. When a control transfer instruction is encountered, recursive traversal continues disassembling at all possible successor instructions. In the case of an unconditional jump, disassembly continues at the jump target; for conditional branches, disassem-

bly continues at the target as well as the fall-through instruction. The benefit of recursive traversal over linear sweep is that it cannot mistakenly disassemble data bytes as code; hence its output is always correct. However, because disassembly stops at indirect control transfers, its code coverage is limited. In that sense, it sacrifices coverage for guaranteed correctness while rewriting.

2.2.3 Pattern Matching

A variety of techniques employ pattern matching to identify bytes such as the bounds of jump tables or to identify function prologues. Typically, these techniques are architecture- and compiler-specific, which is a drawback since binary codes can be produced by a wide variety of compilers and by hand-writing assembly code. Further, these techniques cannot guarantee correctness since a series of data bytes might also coincidentally fit the target pattern; hence we do not use this technique.

2.2.4 Speculative Disassembly

Another method used to increase code coverage is speculative disassembly. It recognizes portions of the code segment that have not yet been disassembled and assumes that these gaps in the disassembly are most likely the targets of indirect control transfers. Disassembly is then restarted at the beginning of these identified blocks assuming they are code. If disassembly encounters bit patterns that are not legal instructions, then we know that those blocks must have been data and must not be rewritten. However, unfortunately, the opposite is not true: a block that is

actually data may coincidentally also look like legal instructions. Rewriting those would break the code. Hence speculative disassembly is normally unacceptable for rewriting since it could lead to incorrect code. Consequently, existing rewriters do not use speculative techniques, but pay the price in less than 100% code coverage.

2.3 Troublesome Features in Benign Programs

We list some troublesome features that may occur in benign programs. These must be handled correctly since our goal is robust binary rewriting. Existing static rewriters have low overhead, but do not handle any of these features in general.

2.3.1 Obfuscation

Obfuscation is a technique used to mislead attempts to reverse-engineer the code. Here we are primarily concerned about control-flow obfuscation, which makes it appear that data is code, or vice-versa. (We are not concerned with symbol obfuscation which makes the program harder to read by a human by changing symbol names. Symbol obfuscation does not affect most binaries, since stripped binaries lack symbol names anyway.) However, control-flow obfuscation is relevant for binaries. There are publicly available applications and research methods which will control-flow-obfuscate a binary application to further protect the binary from reverse engineering, such as the binary obfuscation project tool [43], the Arxan tool [44], and the work by Popov et. al [45].

There are two types of obfuscation techniques that are problematic for binary

rewriters, and hence are discussed here: (i) Unconditional to conditional branch flow obfuscation, and (ii) Exception-based obfuscation. Both rely on tricking a disassembly routines like recursive traversal to make data appear to be code. Relying on this for rewriting could break the program.

In the unconditional to conditional branch flow obfuscation, an unconditional branch is replaced by a conditional branch, whose one target is never taken. Instead the never-taken path contains data. When recursive traversal is used in a rewriter with this obfuscation, it will falsely assume both targets are code and both will be disassembled and instrumented. This will modify data incorrectly as code, thus breaking the program.

Another technique is exception-based obfuscation. In this technique, a change of control flow is achieved without a control-transfer instruction (CTI), using exceptions instead. For example, the program may contain a DIVIDE instruction in which the programmer (or obfuscation tool) deliberately triggers an exception by using a zero value in the denominator. The program also registers a custom exception handler for divide-by-zero exceptions, whose code can jump to any target instruction in the program. In this way, a DIVIDE instruction can act as a jump that disassembly cannot track. A binary rewriter relying on such disassembly may miss the target code, thus hurting coverage; and may incorrectly assume the bytes after the DIVIDE are code, when they could be data, thus breaking correctness. There are several other exception types that can be used.

2.3.2 Dynamically Generated Code

Dynamically generated code is actually common in benign applications. It is mostly used when executing user scripts or any script coming from external sources. Another reason for having dynamically generated code is packed code, used in a few benign programs for obfuscation. In this approach, the data in the code segment is unpacked during the execution of the program and then control is transferred to this newly generated code using some sort of CTI. This obfuscates code to look like data, so that human reverse engineers or static disassemblers miss that part of code.

Unlike dynamic rewriters, static rewriters cannot disassemble such dynamically generated code. Our binary rewriter ensures that the newly generated code is fully disassembled and instrumented, without incurring high overhead.

2.3.3 Self-Modifying Code

Self-modifying code is similar to dynamically generated code with an important difference: the addresses into which dynamically generated code are stored may already contain instructions that have been executed during the program. This modifies the program's code at run time.

Self-modifying code could cause serious problems for all types of binary rewriters, since it means that all the assumptions about the instructions in those memory addresses are no longer valid after self-modification. For example, a problem could be that all the instrumentations that were in those memory addresses, are now lost because they are rewritten by the newly generated code. In the later chaoters, we

will describe our proposed mechanism for RL-Bin to handle these cases.

2.4 Limitations of RL-Bin and Introduction of RL-Bin++

RL-Bin is capable of analyzing and instrumenting most of the common commercial binary files which do not have relocation information, and may have obfuscated, dynamically-generated or self-modifying code. However, RL-Bin is not designed to support adversarial binaries, which can deliberately use methods to prevent their examination by a binary rewriter or a debugger.

Here are certain types of behavior in adversarial binaries that can cause problems for the binary rewriter.

2.4.1 Verifying the Memory Image Checksum.

Some adversarial binaries compare the checksum on their memory image against a previously calculated checksum to make sure that the program is not altered by debuggers. The goal is not ensuring integrity, but defeating debuggers. In most commercial binaries, developers know that many users may use debuggers on the software which will not work with such binaries.

2.4.2 Disabling the Debugger.

Binaries can check the presence of a debugger, and if found, can try to disable it. As mentioned before, commercial binary applications support debuggers.

2.4.3 Debugger Breakpoints Modification.

Adversarial binaries attempt to remove breakpoints inserted by debuggers, which can interfere with the operation of rewriters and debuggers. This behavior is limited to only adversarial binaries.

Due to these limitations, RL-Bin might not properly instrument binaries with aformentioned features. Hence, we have introduced RL-Bin++ in Chapter 6 which is capable of handling troublesome features. These features are desribed in detail in Chapter 5

Chapter 3: Design

In this chapter, we describe the base un-optimized algorithm that is used by RL-Bin. This algorithm has very high overhead (approximately 5x to 10x the run-time of the un-instrumented program for SPEC CPU2017 benchmarks) but demonstrates the correctness of the method.

3.1 System Design Overview

The components of RL-Bin are shown in Figure 3.1. The Control Unit keeps the state of the application and manages other units. The Instrumentation Unit creates and manages instrumentation routines. The Trampoline Unit is responsible for efficiently placing trampolines in the original code to redirect execution to the instrumentation routines. Finally, the analyzer and optimizer unit is responsible for optimizing and removing instrumentation routines that are no longer needed.

3.2 RL-Bin Baseline Algorithm

Figure 3.2 shows intuitively how RL-Bin discovers and executes code. The main intuition behind RL-Bin is to add instrumentation at run-time that monitors the discovery of the new code. To discover code, our method assumes that a block of

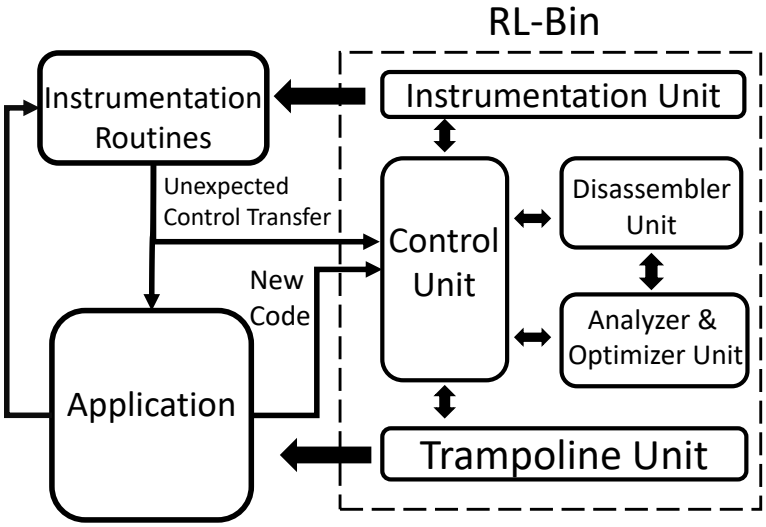


Figure 3.1: RL-Bin System Overview

memory is code only if we discover an actual control transfer to it during run-time. Our **purely dynamic** disassembly method will begin at the start of a memory block (whose address we call START) once it is proven to be code and follows non-control-transfer instructions one after another, which are all discovered to be code, until it reaches a control transfer instruction. Whenever the method reaches a CTI, if that CTI can have more than one possible target, the method ensures that some instrumentation is triggered when the actual target becomes known later during the same run.

Some terminology: All instructions that change the control-flow behavior of a program, such as branches, jumps, and calls, are called *Control-Transfer Instructions (CTIs)*. A *direct CTI* is a CTI whose target is specified by an immediate constant in the instruction. Direct CTIs can be unconditional or conditional. An *indirect CTI* is a CTI whose target is specified in a register or memory location and hence is usually unknown statically.

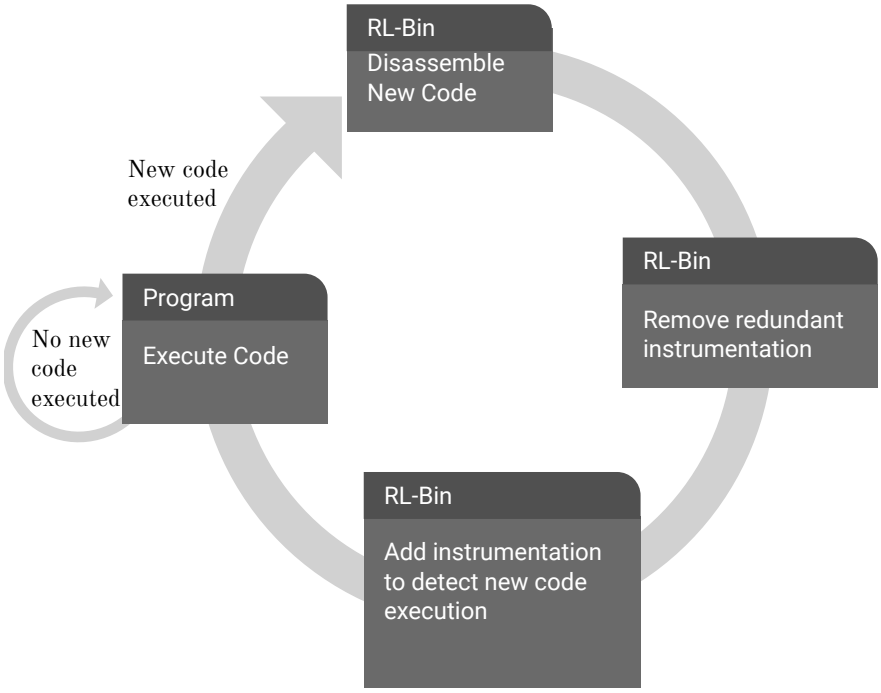


Figure 3.2: RL-Bin's Cycle of Code Discovery and Execution

Here are the steps in RL-Bin dynamic **Disassembly Routine:**

1. Add entry point to the list of instructions to be discovered, let's name it D.
2. Pick an instruction I from list D.
3. Mark the address of instruction I as discovered in the disassembly table.
4. If instruction I is a non-control-transfer instruction,
   (a) The next instruction must be code as well, so we add it to list D if has not been disassembled before.
5. If instruction I is an unconditional direct CTI,

(a) It has only one possible constant target (*i.e.*, it is a direct jump), so we can infer that the target is definitely code as well, so we add the target to list D and disassembly continues from there.

6. If instruction I is a conditional direct branch, (see Figure 3.3 as an example)

   (a) We cannot assume that its target (T) and fall-through (F) addresses are both code. As discussed before in section 2.3, because of conditional branch obfuscation, only one of the target or the fall through may be code, but not necessarily both. Hence we insert hardware breakpoints at both the target and fall-through addresses (T and F).

   (b) Register a custom exception handler for handling these hardware breakpoints. Particularly, when either one of them is executed (say T),

      i. It will register that memory location as code in the disassembly table.

      ii. Then it removes hardware breakpoints at both T and F. (The reason that hardware breakpoints are removed from a block after it is executed is that in most ISAs, only a small number of hardware breakpoints is allowed at a time. In the case of x86, there is a limit of four hardware breakpoints that can be set at a time.)

      iii. Adds trampoline at START (see trampoline (1) in Figure 3.3), which will transfer to instrumentation routine that adds back the hardware breakpoint at the non-executed address among T and F (say F) (If the code is executed from START again, we do not need to disassemble the code from START again, but just insert the hardware

breakpoint at F when at START.)

Note: In the case of x86, if there are more than four non-executed addresses in the function, extra trampoline(s) will be placed in the middle of function to remove hardware breakpoints from previous addresses and insert them on the following addresses.

(c) Later, as an optimization, if the handler at F also executes, remove the hardware breakpoint, as well as the instrumentation at START. This leads to zero overhead in the steady state after T and F are both proven to be code.

7. If instruction I is an indirect CTI,

   (a) Insert trampolines to an instrumentation routine (see trampoline (2) in Figure 3.3), just before the indirect CTI to the instrumentation routine which,

      i. Computes the target upon reaching that point.

      ii. Add it to the list D, if it is not disassembled before.

   (The target of indirect CTIs need to be checked every time because it can change every time the instruction is executed; hence, our trampoline and instrumentation will remain in place to check the target of indirect CTIs to discover new code and handle unexpected control flows.)

8. If D is empty, then exit, otherwise go to step 2.

The above method works for dynamically-generated code without a special

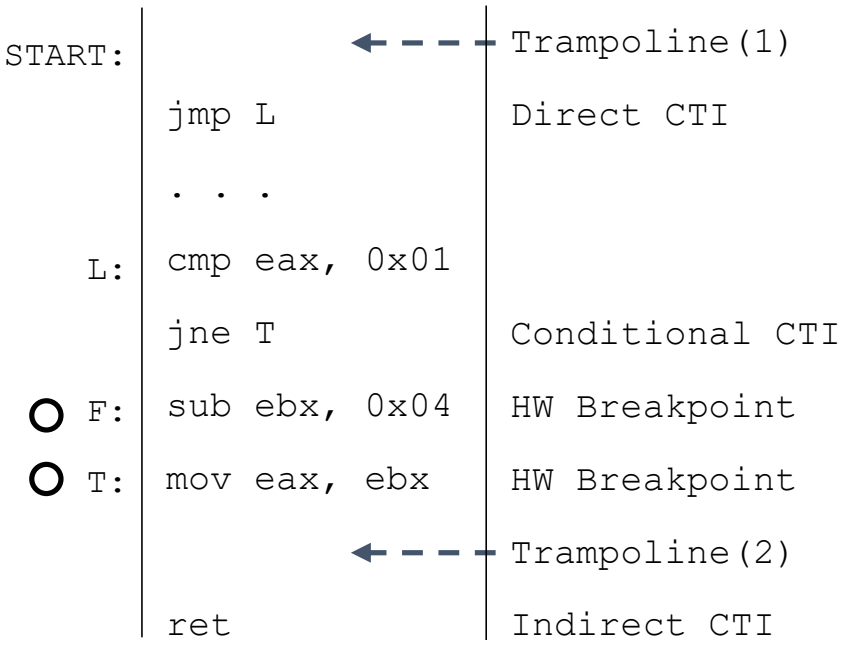


Figure 3.3: Disassembling a Memory Block

case, since it tracks the CTI into the dynamically-generated code just like any other CTI. It also handles unconditional to conditional branch obfuscation as described above. However, the method needs additional components to handle self-modifying code and exception-based obfuscation. These will be described in subsections below.

3.3 Handling Self-Modifying Code

Self-modifying code is handled as follows.

1. To check whether the code has modified itself, write-protect the pages that contain code, so any write to these pages will cause an exception.

2. Register the exception handler to:

   (a) Check the addresses which are being written.

(b) If they have previously been discovered as code, remove those entries from the disassembly table. (As a result, the newly written code will be treated the same as the code which has never been seen before.)

The above method is very high overhead and it needs to be optimized. The main overhead comes from the fact that every write to the code segment will cause an exception. Such writes will happen if data is stored in the code segment and is written to by the program. To reduce the overhead, we use the following scheme. We add instrumentation code around memory store instructions that trigger the exception for the first time. The instrumentation will turn off write protection, check the addresses being written to, and turn back on write protection after the memory store. In this way, stores to data locations in the code segment will never trigger an exception more than once. As a result, only a small portion of memory store instructions (those that write to the code segment) will be surrounded by our added instrumentation.

3.4 Handling Exception-Based Obfuscation

This obfuscation happens when an instruction that is not a CTI is used to transfer control of the program. As an example, a divide instruction which deliberately triggers an exception can be used as a CTI. As a result, the memory location following the divide instruction may never be executed. Actually, it may contain data and not code. To handle exception-based obfuscation, we follow the following method.

1. Create a stub for every exception handler that is registered. When an instruction triggers the exception it will execute our instrumentation before the actual exception handler.

2. Disassembly routine must stop disassembly at every instruction that can cause an exception that has been registered so far. (In the common case no such exceptions will be registered, thus the overhead will be minimal.)

   (a) If such an exception causing instruction is found (in step 4 of the baseline algorithm), put a hardware break-point on the instruction that immediately follows it.

   (b) After hitting the breakpoint, remove it and start discovering code from that location. (This method ensures that no data is mistakenly assumed to be code.)

Using the algorithm in this section, more and more code is discovered during run-time. This method will ensure that not a single instruction can be executed without first being observed by our binary rewriter, even if the instruction has been generated dynamically or through self-modification. Also, in case there is obfuscation, we would never instrument data inside the code segment since we instrument only the locations that contain code that has been executed during run-time.

3.5 Handling Multi-Threaded Applications

By the advent of multi-core processors, multi-threaded applications have become very common. As a result, every binary rewriter must handle such applications. The main issue in multi-threading is to make sure that the data structures that are shared between threads are being used correctly. Specifically, they should not be used by a thread while simultaneously being updated by another thread. To avoid the problems regarding concurrent access to RL-Bin data structures, each thread must acquire the lock before being able to modify RL-Bin internal data structures. During this modification, no other threads are allowed to access the same data structure.

3.6 Disassembly Table Structure

Here we describe the disassembly table, which is the main data structure used in the algorithm. The table keeps essential information about each byte of the code segment in the main memory. Each entry of the disassembly table is used to track the status of the corresponding memory location, and when a new action needs to be taken.

For each byte in the code segment, there is a two-bit entry in the table which can show one of four different possibilities (0 to 3) as the status code for that location. If there is no information about the byte, whether it's code or data, then the entry is set to 0. If the byte is code and also it is the start of a basic block, the

entry is 1. If the byte is code and but not the start of a basic block, the entry is 2. If the byte has been modified by the rewriter to add instrumentation, the entry is 3. Figure 3.4 shows an example of a disassembly table. In the figure, the status codes for 0, 1, 2, 3 are encoded in binary in the expected way as 00, 01, 10 and 11, respectively.

3.7 Instrumentation and Trampolines

Since RL-Bin is an in-place binary rewriter, a trampoline is used to insert any instrumentation in the original code. A trampoline is an instruction that we insert,

Corresponding Memory Location	2-bit Entry
...	...
0x10000039	00 (Unknown)
0x10000040	01 (Code & Start of BB)
0x10000041	10 (Code)
0x10000042	10 (Code)
0x10000043	10 (Code)
0x10000044	10 (Code)
0x10000045	11 (Instrumentation)
0x10000046	11 (instrumentation)
...	...

Figure 3.4: The Disassembly Table

which rewrites an original known instruction in the code segment, and replaces it by a CTI that will redirect control to the instrumentation of our choice that needs to be executed at that point. The instrumentation itself is added elsewhere in the binary, typically at its end. The instrumentation includes at its beginning the instruction that was overwritten, followed by the new code of our choice to be inserted, then followed by a CTI at its end that redirects control back to the original code at the instruction following the overwritten instruction. Trampolining is a well-known way of inserting code in binary rewriters.

There are several options for choosing the instruction used as the trampoline. Each of these options could be used depending on the instruction inside the basic block that needs to be instrumented.

In the best case, a five-byte long jump can be used to divert the execution to the instrumentation. This is the preferred method since it is very low overhead and also it would allow the instrumentation to be in any location in the main memory. The only downside is to find one or more instructions with total length of five bytes to be replaced by the trampoline.

Another choice is to use a two- or three-byte short jump. The advantage is that finding the instruction to be replaced is easier; however, since it is a short jump, there is not much flexibility regarding the location of instrumentation. This issue could be resolved in two steps: first by jumping to a five byte long jump, and from there trampolining to the actual instrumentation which could be placed anywhere in the memory.

The least efficient way is using the one-byte trap. Since, it takes only one byte,

it could be inserted anywhere that is needed. This trampoline is used as the last option, if the methods described above cannot be used. The reason is that a trap instruction takes hundreds of cycles to be executed and using it frequently would lead to a very high overhead. Trampolining itself is well understood and common, so many of the policies for optimizing trampolines are not new, and have been quite successful in the past.

Chapter 4: Optimizations

This chapter presents the optimization techniques used in RL-Bin to reduce the overhead. The effectiveness of each optimization will be discussed in Section 7.3.

4.1 OP1. Conditional Branches

As was described in step 6.3 of the baseline algorithm, if at any point both outcomes of a conditional branch are registered as code, then the instrumentation and hardware breakpoints at that branch can both be removed. In the steady state, the checks before most direct conditional branches are removed.

4.2 OP2. Predicting the Target of Indirect CTIs

The baseline algorithm in step 7, instruments every indirect CTI to compute its target at run-time and register it as code if it is the first time that the address is executed as code. This overhead can be reduced by an optimization with the following intuition: indirect CTIs usually transfer the control to one of a few constant targets. We will put a check which takes less time before this heavy check of an indirect CTI.

As an example, let's assume that function *foo()* is being called from three different call sites. So, the return instruction of the function will return to the instruction after one of these call sites. First, the target will be checked against the most frequent call site. If it matched, the indirect CTI can safely transfer the control flow back to the call site. The same idea would be done for second and third call sites. In the end, if none of the previous checks were true, we would refer to the disassembly table to check whether the target of indirect CTI has been discovered as code before.

4.2.1 Discussion on the trade-offs between different methods of prediction

We will discuss two different methods that we tested for the prediction of indirect CTIs. The first method is to use profiling to find out the most frequent destinations of an indirect CTI. Based on the profiling information, we decide the number of predictions for each branch. The other method uses a fixed number of predictions and does not rely on profiling information. In this method, we dynamically change the prediction if it was not correct. We discuss the two methods in detail.

In the first method, the number of predictions will be determined based on the frequency of the times that a particular destination is taken. The heavy check (compute the address and check if it has been discovered before) in our implementation costs around 40 cycles. The light check that compares the target against a

constant value costs 11 cycles in our implementation. If a light check is not taken, we continue with the rest of the predictions, and if all of them are incorrect, we fall back to the heavy check. Formula 1 shows the total cost of the mentioned checks. Here H is the cost of the heavy check, and L is the cost of light checks. Also, f_1 to f_n are the frequency of the destinations based on profiling information. N is the number of destinations for the indirect CTI. Also, i is the counter to help demonstrating the cost of the check for the i-th destination.

(1) Total Cost of Checks for an Indirect CTI $= \sum_{i=1}^{N} i f_i L + (1 - \sum_{i=1}^{N} f_i)(NL + H)$

In order to find out the optimal number of light checks for each indirect CTI, we include the first K frequent destinations out of N total destinations. As an example, consider an indirect branch with the frequency of target addresses 0.8, 0.1, 0.02, 0.01, etc. For one light prediction, K=1, the total cost is (0.8*11+0.2*51 = 19). The cost is (0.8*11+0.1*22+0.1*62=17.2) for the prediction of two targets, K=2. For three targets, K=3, the cost would be (0.8*11+0.1*22+0.02*33+0.08*73 = 17.5), which means that having two predictions (K=2) is the optimal case.

In general, to calculate the total cost of checks for K predictions, we replace K for N in the formula above. We continue increasing the number of predictions, K, until total cost reaches a minimum value, after which increasing K increases the total cost.

Another method of prediction uses a purely dynamic method based on the

intuition that a branch is likely to take a target if it has been taken recently. We have implemented and tested two versions of this method, with one or two predictions. In the first version, we predict only one target and use light check only for that prediction. Our prediction is the last destination that is taken by the indirect CTI. If the predicted address is wrong, we change the prediction and replace it with the newly taken destination. In the second version, two addresses are predicted. If we have a misprediction, we replace the first prediction with the second prediction, and the second prediction will be replaced by the last destination that is taken by the indirect CTI. This method has more memory overhead, but its run-time overhead is less than the first version.

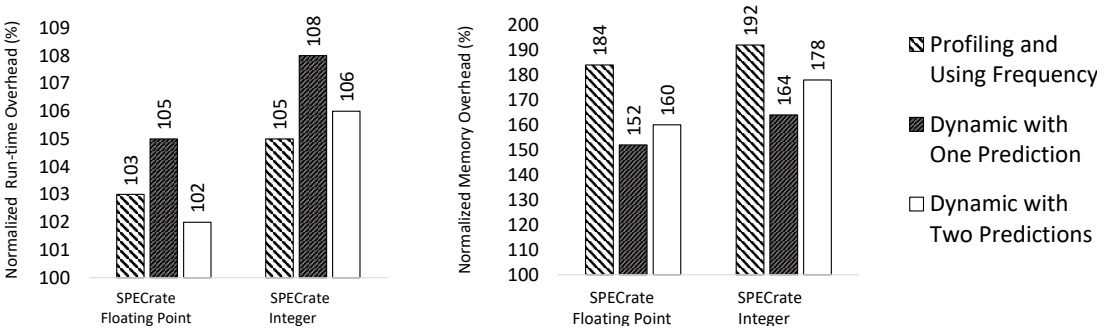


Figure 4.1: Run-time and Memory Overhead of Applications with Three Methods of Prediction

Figure 4.1 shows the memory and Run-time overhead of SPECrate 2017 benchmark applications with three prediction methods: (1) profiling; (2) dynamic with one prediction; and (3) dynamic with two predictions. The numbers shown in this figure assume that all other optimization methods of RL-Bin have been applied. As can be seen, the run-time overhead of the dynamic method with two predictions is comparable with the profiling method, but its memory overhead is significantly less.

The dynamic method with one prediction has the lowest memory overhead, which comes at the expense of higher run-time overhead. Depending on the use case, one of these methods can be used. The dynamic method with two predictions seems to fit most of the cases because of the balance between its memory and run-time overhead and the fact that it does not rely on profiling.

4.3 OP3. Function Cloning

It is often the case in programs that a small function is directly called frequently from a call site. The intuition is to remove the check needed before the return instruction (indirect CTI) to the call site. During the step 7 of the baseline algorithm, we selectively clone functions to reduce the overhead and remove the checks needed before their return instructions.

In this method, the function is cloned so that no check is needed if called from that specific call site. First, the function is copied to a new location. The call instruction is modified to a direct jump to the new location. As a result, no return address will be pushed on the stack. Also, the return instructions in the function are replaced by direct jumps to the instruction after the call site.

4.3.1 Discussion on when to perform function cloning

Function cloning has the advantage of removing indirect control flow instructions and the checks needed. However, the downside is that the cloning process itself is not free and has high overhead in terms of run-time and memory. Thus, we

need to decide when it is appropriate to clone a function for a particular call-site.

First, we will solely focus on run-time overhead. The ongoing cost is to execute the return instruction and its instrumentation. If we decide to perform function cloning for a call site, we pay a higher one-time cost of cloning, but we would no longer have the ongoing cost. If we already knew how many times a particular call-site would call the function throughout the execution, it would be easy to calculate whether it is cost-effective to perform the cloning.

The uncertainty of the number of execution times of the call instruction is similar to the snoop caching problem. According to [46], when there is an ongoing cost and a higher cost to get rid of the ongoing cost, the efficient choice is to pay the one-time cost after the sum of ongoing costs by that point exceeds the former. As a result, we should clone the function after it has been executed enough times so that the cost of return and its instrumentation exceeds the cost of cloning. Hence, the formula for finding the threshold number is the following, where C_c is the cost of cloning the function per byte, S is the size of the function in bytes, and C_r is the cost of running the return and instrumentation.

$$T = \frac{C_c S}{C_r} \tag{2}$$

Figure 4.2 illustrates the run-time and memory overhead when function cloning is applied at higher thresholds. The overhead numbers in this figure are measured after all other optimizations of RL-Bin have been applied. The numbers show that as

the threshold increases, the run-time overhead increases, but the memory overhead goes down. If the goal is to achieve the lowest possible run-time overhead, cloning should be done at the threshold T. Otherwise, one can increase the threshold to have a smaller memory footprint. By default RL-Bin is configured to apply function cloning at the threshold T. If the goal is smaller memory overhead, then the threshold 10T seems to be a good choice, because it has only 31% memory overhead and its run-time overhead is 7% which is an acceptable overhead for deployment in live systems.

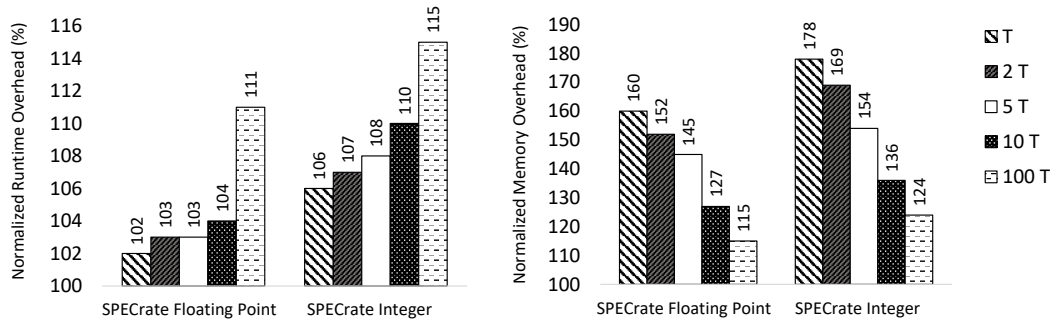


Figure 4.2: The Trade-off Between Memory and Run-time Overheads With Different Thresholds

4.4 OP4. Optimizing Whitelisted Modules

It often happens that applications load dynamically shared libraries during their execution and then execute functions from them. In most cases, these DLLs are part of the kernel or they are part of the standard library provided by the programming language. It is possible to optimize away the checks needed for some of these DLLs.

The interaction between the main module of the program and the shared

libraries happens by calling a function exported by the library. The control will be sent back to the main module after the execution of the function. The only exception is when the library performs a callback and calls a function from the main module. DLLs are analyzed and their callback functions are discovered. If the behavior of the functions and the callback values can be determined prior to execution, then the analyzed DLL will be whitelisted and checks in that module will be optimized away.

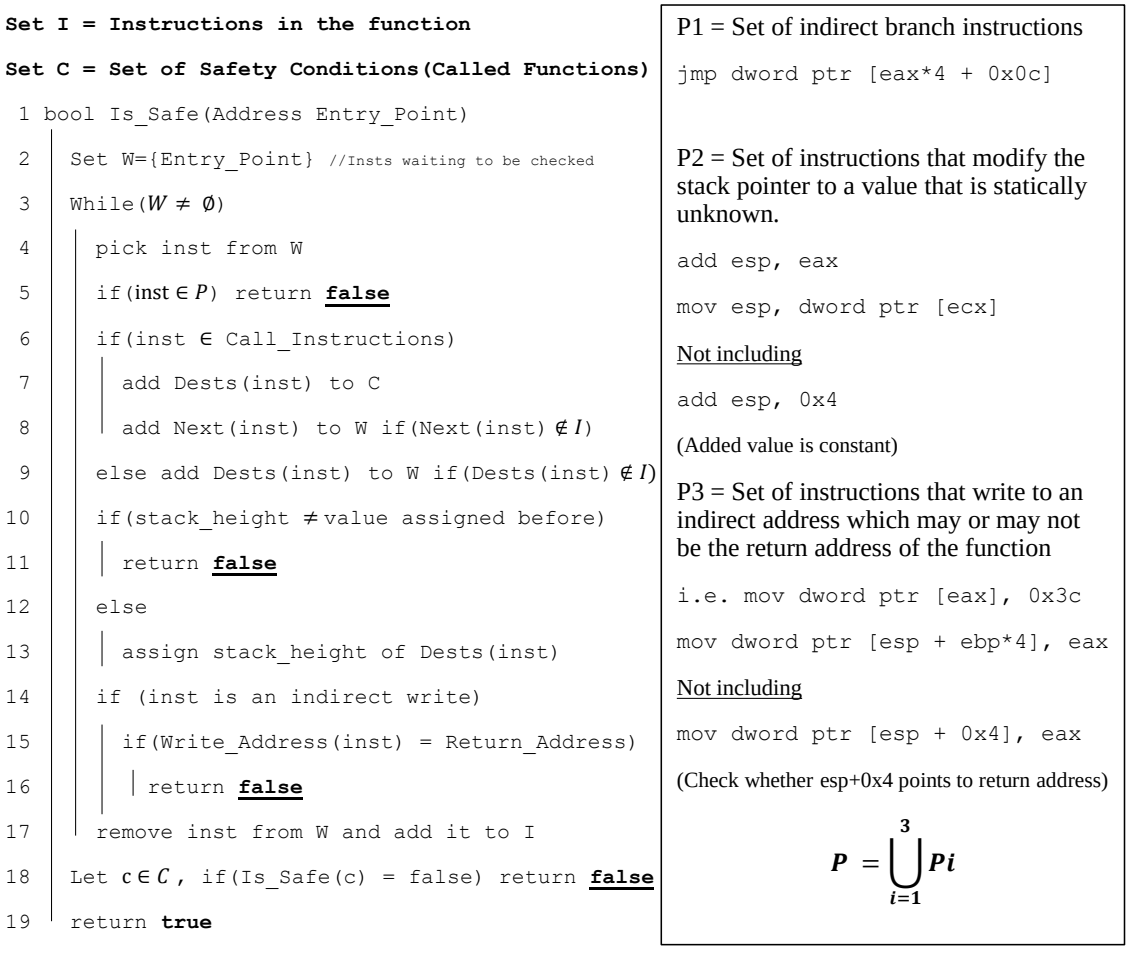


Figure 4.3: The Algorithm to Determine Safety of a Given Function. (None of the instructions in set P, defined on the right side, are allowed in a "Safe" function. Dests(inst) return the targets of CTIs and for non-CTIs, returns the next instruction.)

4.5 OP5. Detecting "Safe" Functions

The most common indirect CTIs are return instructions. The overhead of the checks before return instructions, checks added during step 7 of the baseline algorithm, can be further eliminated when the function has certain properties. A "Safe" function, can be proven that it cannot modify its own return address, hence the return instruction always returns to the instruction after the call site.

We outline in Figure 4.3, our Just-In-Time (JIT) analysis algorithms, by which the safety of many functions can be established before their execution. For such safe functions, the instrumentation before the return instruction can be removed. The intuition behind the algorithm is to determine the exact addresses of the memory locations on the stack that will be modified by the instructions within the function. If the return address is not modified, then the function will return to the original call site.

"Stack Height" for every instruction, is defined to be the difference between the value of the stack pointer at the entry point of the function and the value of stack pointer at that instruction. For example, a push instruction will reduce the "Stack Height" by four. If the function does not contain any of the instructions defined in Figure 4.3 as set P, the "Stack Height" of all instructions can be determined prior to the execution of the function. If there is more than one control flow paths from the entry point to a given address, and "Stack Height" is not the same between different paths, we declare that function as not "Safe" and do not optimize it. This rarely happens in benign code.

The algorithm will determine the "Stack Height" of each instruction and based on the "Stack Height", will determine whether an indirect write rewrites the return address of the function. We also create a list of functions that are called from this function and put them in set C. Later on, after disassembling all instructions in the function, we check the safety of all the functions in set C. If any of the called functions is not safe, the current function will be declared not "Safe". If all the aforementioned checks showed that the return address cannot be modified, the function will be declared "Safe". Note that the algorithm above will be executed only once for each discovered function, thus there will be no overhead in the steady state.

4.6 OP6. Using Data-Flow Analysis to Find "Safe" Functions

OP5 algorithm does not cover some functions, because writing to global or static data, which is not stored on the stack, is frequently done through indirect addressing.

If there is a write to an indirect address, we need to make sure it does not overwrite the return address of the function. Most of the indirect writes to the stack are done using stack-derived registers as base registers (In x86, these are esp and ebp registers). So, if the base register is not a stack-derived register or it's not a copy of these registers, then it cannot modify any value which is previously stored on the stack. As a result, we must ensure the base register is not derived from the stack pointer.

We define the term PNSD, which is short for "Provably Not Stack Derived". If a register value is PNSD, it means that it can be proved during run-time analysis that the current value in the register is not derived from the stack pointer. An indirect write instruction which uses a PNSD register can never write to the stack. We use traditional data-flow analysis to identify all the different definitions that can reach the base register in the write instruction. If all of the definitions of the base register are PNSD, then the base register is also PNSD.

As it is demonstrated in Figure 4.4, we modify the algorithm in the previous section to check for PNSD variables when there is an instruction, which stores the value to an indirect address. Again, note that the analysis above will be done only once for each discovered function, thus there will be no overhead in the steady state.

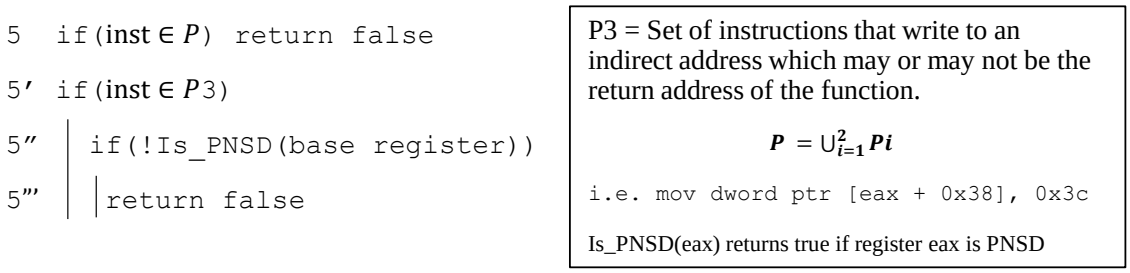


Figure 4.4: Algorithm Modification to Cover Indirect Writes with PNSD Base Register.

4.6.1 Discussion on when to apply OP5 and OP6

Optimization methods 5 and 6 provide the advantage of removing the check before return instruction of safe functions. However, doing just-in-time analysis when the code is discovered is very costly. The optimization will only be beneficial if the cost of the analysis to determine the safety is amortized by the elimination of

instrumentation in multiple executions of that function.

Similar to the discussion above made for function cloning in subsection 4.3, according to [46] it would only be beneficial to apply the analysis when the function has been executed enough times that the overhead is more than the cost of analysis. Based on the following formula, we determine the threshold at which we would apply the analysis for optimizations 5 and 6, where C_5 and C_6 are the constants related to average cost of these optimizations per byte, and S is the size of the function in bytes. Note that the data flow analysis in optimization 6 has quadratic time complexity in S.

$$T_5 = \frac{C_5 S}{C_r} \tag{3}$$

$$T_6 = \frac{C_6 S^2}{C_r} \tag{4}$$

Figure 4.5 demonstrates the percentage of functions in the SPECrate 2017 benchmark, the safety of which can be determined by optimizations 5 and 6. We have also shown the percentage of amortizable safe functions, the subset of Safe functions for which it is worth it to analyze them to find if they are safe. For amortizable safe functions, the benefit in run-time overhead is more than the time of analysis. The analysis is done only once per function but every time the function

is executed we get the benefit of not instrumenting it. As it can be seen in this figure, to get the lowest run-time overhead, optimizations OP5 and OP6 should only be applied to the functions that are executed enough times to amortize the analysis cost.

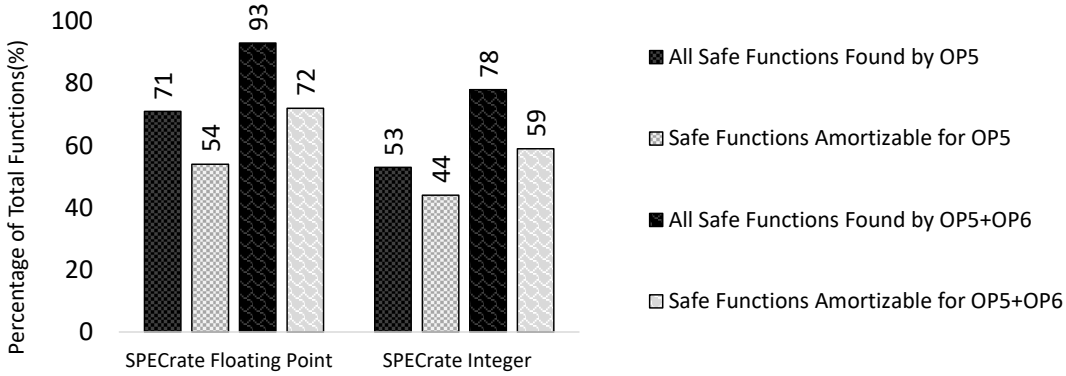


Figure 4.5: Percentage of all Safe Functions and Amortizable Safe Functions For OP5 and OP6

Similar to the discussion in function cloning in subsection 4.3, we can apply OP5 and OP6 optimizations at a higher threshold. That means that the function must be executed several times more than the analysis time. Unlike function cloning, this does not lead to a trade-off between memory and run-time overheads. Applying OP5 and OP6 at a higher threshold would lead to both higher run-time and memory overhead.

We also considered applying OP5 and OP6 at a lower threshold, meaning that OP5 and OP6 are applied to functions that are not executed enough times that would amortize the function analysis time. This would lead to higher run-time overhead but reduces the memory overhead, because proving that a function is safe means there is no need to instrument the function. Although memory overhead is reduced,

it is not by a very significant amount. In our experimental results for SPECrate 2017 applications, if we perform OP5 and OP6 analysis for every function (which are not necessarily amortizable), memory overhead is reduced form 69% to 60% and run-time overhead increases from 4% to 9%. The memory overhead reduction is not worth the increase in run-time overhead, hence we apply OP5 and OP6 only when the threshold is reached for each optimization.

Chapter 5: Difficulties of Obfuscation for Binary Rewriting

Obfuscation refers to all different techniques used to conceal the structure of the code. These methods were previously frequently used by malicious software, but recently, it has become more common in benign applications. The reason is that commercial applications need to be protected against reverse engineering. In some cases, obfuscation would help with digital rights management (DRM) and avoid undesired or illegal software uses.

Obfuscated binaries have complex features that can be categorized into the following groups: Anti-Disassembly, Anti-Debugging, Anti-Rewriting, Dynamic Code, and Convention Infringement. In general, these features are added to prevent the software from being reverse engineered or modified. The problematic characteristics of obfuscated binaries intentionally mislead analysis tools such as disassemblers, debuggers, and rewriters.

5.1 Explanation of Problematic Features

Several features are introduced by obfuscators to throw off analysis tools. This section categorizes these features based on the similarity and the targeted analysis tool. Table 5.1 shows the five categories of these challenging features. This section

describes the reasoning behind each type and how they can affect analysis tools.

Categories	Problematic features
Anti-Disassembly	Obfuscating CTI Targets Ambiguous Code and Data
Dynamic Code Modification	Dynamic Code Unpacking Self-Modifying Code
Anti-Rewriting	Self-Checksum of code
Convention Infringement	Section Label Misuse Function Handling Obstruction Calling Convention Misuse
Anti-Debugging	Debugger Resistant

Table 5.1: Categories of Problematic Features Found in Obfuscated Binaries

5.1.1 Anti-Disassembly

Disassembly tools such as IDA Pro [47] depend on extracting instruction from the binary file by well-known methods like linear sweep or recursive traversal. Linear sweep starts at the entry point of the binary and consecutively disassemble instructions one after another. On the other hand, recursive traversal begins at the entry point and follows control transfer instructions to disassemble code.

There are three primary methods of throwing off disassemblers. The first method is creating unnecessary indirection in CTIs, whose target is unknown before

execution and will be missed by static disassemblers. The second method is creating ambiguity by converting unconditional branches to conditional branches with two targets. One of the targets is fake and contains data instead of code. Disassemblers using the recursive traversal method would mistakenly disassemble invalid instructions from the fake target address. The third method is changing the control-flow by deliberately causing exceptions during run-time. Static tools would not realize that exception would happen during run-time, and therefore they would miss the program's original control-flow.

5.1.2 Dynamic Code Modification

Most of the analysis tools are static. Hence, it is enticing for obfuscation tools to generate and execute code dynamically. This code will remain hidden from all static tools. We categorize dynamic code modification to dynamically-generated and self-modifying code. The difference between dynamically-generated and self-modifying code is that the latter replaces the code that has been executed before. The most common method used by obfuscators is unpacking encrypted code and running the decrypted payload.

5.1.3 Anti-Rewriting

Anti-rewriting techniques are used to ensure that the binary is not modified by rewriting tools. The technique validates that the original code is not altered by calculating checksum and verifying it against a known value. The checksum might

be calculated over the static binary file or part of the program's memory image. It could be the whole code segment or just the unpacked payload. Another variation of anti-rewriting methods is to hide code in strange locations such as the Portable Executable (PE) header file. The goal is to detect if a rewriter has modified or renamed sections of the binary file. In that case, the hidden code is altered, and the rewriter will be detected.

5.1.4 Convention Infringement

There exist unwritten rules about binary code that is generated by compilers. For example, functions are generally a continuous part of code addresses starting at the entry-point and ending with one or more return instructions. Functions regularly do not share code blocks and follow a calling convention for using registers, passing, and cleaning up the arguments. In addition, memory sections are divided into code and data sections with separate read/write/execute permissions. None of the rules mentioned above is mandatory, and obfuscation tools do not follow them to evade the analysis tools that depend on them.

5.1.5 Anti-Debugging

Obfuscation tools attempt to bypass debuggers by detecting them and interfering with breakpoints. Debuggers register themselves in the operating system as the target process's debugger to get the first chance to handle the exceptions. In addition, debuggers use one-byte software breakpoints and single-step traps, and

hardware breakpoints to monitor the execution of the application. Obfuscation tools can detect the debugger registration and stop execution if a debugger is found. Additionally, they can insert or remove breakpoints that are intended to be solely used by the debugger. By doing that, the debugger would not be able to behave as expected.

5.2 Effect of Problematic Features on RL-Bin

This section discusses in detail the problematic features in each of the five obfuscation categories. Primarily, we examine the effect of each feature on RL-Bin. Some features significantly increase the overhead of RL-Bin, while others would cause the program to behave unexpectedly under RL-Bin. Table 5.2 demonstrates the code artifacts corresponding to each feature. In this section, we will study the effect of these code constructions on RL-Bin.

5.2.1 CTI Target Obfuscation

Unlike conditional and unconditional direct branches, indirect CTIs have an unknown number of destinations which are not known before run-time. Anti-disassembly schemes convert direct CTIs to indirect CTIs so that static disassemblers cannot follow the program's control-flow. As discussed in the previous section, RL-Bin inserts a dynamic check before indirect CTIs. As a result, an increased number of indirect CTIs would lead to higher overhead when the binary is executed under RL-Bin.

Category	Problematic Feature	Code Artifacts
Anti-Disassembly	CTI Target Obfuscation	High Percentage of Indirect Calls High Percentage of Indirect Jumps
	Ambiguous Code and Data	Conditional Branch Obfuscation
Dynamic Code	Self-Modifying Code	Code Rewriting Overwrite of Executing Function
Anti-Rewriting	Memory Checksumming	Checksum of Code
Convention Infringement	Section Protection Violation	Writeable Data in Code Segment
	Function Handling Obstruction	Function Entry Not First Block Functions Share Blocks of Code Non-standard Calls/Returns
	Calling-Convention Exploitation	Flags Used Across Functions
Anti-Debugging	Breakpoint Manipulation	Reusing HW Or SW Breakpoints

Table 5.2: Code Artifacts of Problematic Features.

5.2.2 Ambiguous Code and Data

Another method that is used by anti-disassembly tools is converting unconditional branches to conditional branches. The main goal of obfuscators is to increase the control-flow's complexity so that disassemblers would be less likely to find the actual execution path. In the case of RL-Bin, it will encounter numerous conditional branches for which one of the targets is never executed as code. That would force RL-Bin to move and replace hardware breakpoints frequently, clearly leading to higher overhead.

5.2.3 Self-modifying Code

Dynamic code generation and execution commonly occur when executing interpreted languages such as JavaScript. RL-Bin has no problem handling dynamically-generated code since the code is discovered dynamically. There is no difference between the dynamically-executed code and static code from the rewriter point of view. However, self-modifying code requires RL-Bin to invalidate the result of its analysis for self-modified addresses. Self-modifying is rare in benign commercial binaries, but obfuscation tools would deliberately transform regular code to self-modifying code to hinder the performance of binary analysis tools.

5.2.4 Memory Checksumming

This method is used to validate the code's integrity and ensure that it has not been modified before execution. The checksum can be either calculated for

the binary file or its memory image. The binary file checksum would detect the transformations done by static rewriters. RL-Bin does not modify the binary file so that the file checksum can be verified. However, RL-Bin inserts instrumentation in place of the original code, which would change the memory image checksum. Consequently, the obfuscation tool would detect the presence of RL-Bin, and the program behavior would change accordingly. Binary rewriters should not cause any change in the program's behavior.

5.2.5 Section Protection Violation

Mixing code and data is a popular strategy in binary obfuscation tools. Traditionally, code and data are stored in separate binary sections with their own read/write/execute permissions. While the code section is read/execute-only, writable data and read-only data are stored in separate sections.

Obfuscation tools do not follow these conventions and place writable data in the code section. Any data write in the code section triggers RL-Bin's detection method for self-modifying code. Repeated writes to the code section would significantly increase the overhead of RL-Bin.

5.2.6 Function Handling Obstruction

Reverse engineering is heavily dependent on detecting and analyzing functions within a binary. A function is the smallest piece of code that performs a meaningful task; therefore, analysis tools employ complex algorithms to detect the functions'

boundaries and properties.

Because of the importance of functions in program analysis, the obfuscation tools try to hide the functions' boundaries. This task is done using an alternative sequence of instructions that are equivalent to call/return instructions. For example, a subsequent push and jump can be used instead of a call. Other function obfuscation techniques that are deployed include creating non-contiguous functions or abnormal functions that share basic blocks.

Together these techniques will hinder the "safe function analysis" used in RL-Bin, hence leading to fewer safe functions detected. Our analysis shows that the number of "safe functions" seen in obfuscated binaries is one-third of the binaries before obfuscation. As a result, the overhead of obfuscated binaries running under RL-Bin is unacceptable.

5.2.7 Calling-Convention Exploitation

Compilers generate code that follows appropriate calling conventions. Each calling convention uses a different standard to handle passing arguments and cleaning them up after the call. Still, the common fact is that the flags register's value is not passed across different functions. RL-Bin utilizes this fact and modifies the flags register value in the instrumentation routine for the return instruction. If the flag register value is live across the return, the program may behave unexpectedly under RL-Bin.

5.2.8 Breakpoint Manipulation

This obfuscation technique uses the software and hardware breakpoints to redirect the control-flow. The tool registers the appropriate handler and inserts the breakpoint in the original code. Once the breakpoint is hit, the control-flow switches to the handler, and the execution continue from there. These breakpoints are supposed to be used only by debuggers. If an application uses these breakpoints, the debugger cannot differentiate between the user's breakpoints and the original ones. Particularly, RL-Bin uses software (SW) and hardware (HW) breakpoints to manage the control-flow. If the program manipulates the breakpoints set by RL-Bin, the rewriter might lose control of the application.

Chapter 6: Overcoming Troublesome Features, Introducing RLBin++

6.1 Proposed Methods for Handling Obfuscation

We have designed and implemented novel methods to handle the challenging features that RL-Bin had not previously handled. As discussed in Chapter 5, RL-Bin can still instrument the program with an unsatisfactory overhead for some features. For the rest of the features, RL-Bin might cause the program behavior to change, which must never happen. Our methods would resolve all the issues that were discussed in Chapter 5. Table 6.1 demonstrates which code artifacts are handled by a given method. In this chapter, we will describe each of our proposed methods in detail, and we would show the effectiveness of these methods in Section 7.5.

6.2 Indirect CTI Deobfuscation

RL-Bin follows the execution of the program dynamically by tracking all control-flow transfer instructions, specifically indirect CTIs. Obfuscation tools convert direct jumps and calls to indirect CTIs, forcing RL-Bin to insert instrumentation before indirect CTIs, effectively leading to higher overhead. We propose a deobfuscation method to change these indirect CTIs back to the original direct jump

#	Method Name	Handled Code Artifacts
I	Indirect CTI Deobfuscation	High Percentage of Indirect Calls High Percentage of Indirect Jumps
II	Hybrid Code-Cache - Write Emulation	Code Rewriting Conditional Branch Obfuscation Writeable Data in Code Segment
III	Shadow Memory - Read Emulation	Checksum of Code
IV	Safe Function Detection Improvement	Function Entry Not First Block Functions Share Blocks of Code Non-standard Calls/Returns Overwrite of Executing Function
V	Flag Register Liveness Analysis	Flags Used Across Functions
VI	Breakpoint Protection	Reusing HW or SW Breakpoints

Table 6.1: Methods Introduced to Efficiently Handle Code Artifacts.

or call. Based on our observation of obfuscated binaries, we detected three main patterns to create indirect CTIs. These obfuscation methods are demonstrated in figure 6.1.

The first method is creating an address table to create indirection. Instead of directly jumping to a location, the obfuscator changes the instruction to an indirect jump going to an address stored in the address table. Our method would detect such indirect CTIs and convert them back to the original direct CTI. Then we make

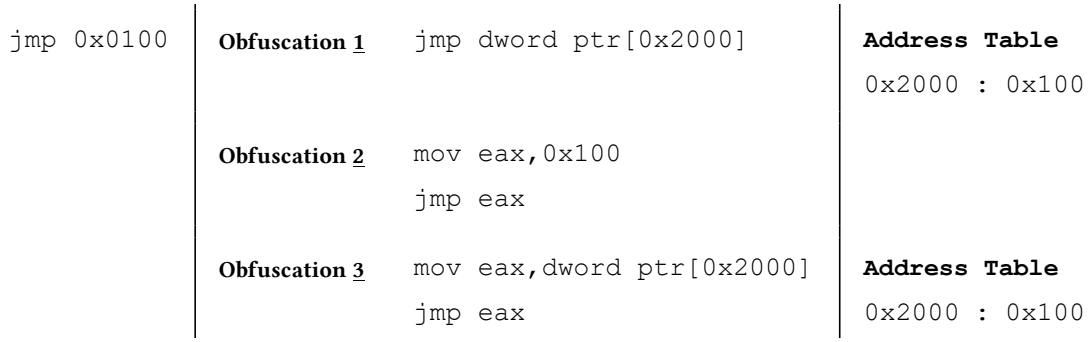


Figure 6.1: Three Patterns of Indirect CTI Obfuscation

sure that the address table remains constant by write protecting it. If the address table values are modified at any later point, we will change the corresponding direct CTIs. The second method of indirect CTI obfuscation is storing a constant value in a register and using that register as an operand. The instruction that stores the constant value might not be immediately before the jump instruction. Our workaround is to perform Just-In-Time (JIT) data-flow analysis on the basic block that ends with the indirect CTI. If the register has only one definition reaching the indirect CTI, we will convert the indirect CTI back to the original direct CTI. The last type of obfuscation in our study is the combination of previous techniques. Our solution would be performing the same data-flow analysis on the operand register. Additionally, we write-protect the address table and update the corresponding CTI if the table is modified.

6.3 Hybrid Code-Cache - Write Emulation

Compiled binaries rarely write a value on code pages. On the contrary, the behavior is frequently observed in obfuscated binaries. A write to a code page might overwrite an instruction, self-modifying code, or write to a data address. We

propose a combination of code-cache and write emulation to handle both cases of self-modifying code and frequent writes to the code section. A code-cache is a copy of code blocks. If a block of code is on the code-cache, the original copy is never executed, and only the copy of the block can be executed from the code-cache. Implementing the code-cache would also help with conditional branch obfuscation, which will be discussed later in Section 6.6.

The disassembly table keeps the information of executed code addresses. If a block of code is copied to the code-cache, the corresponding value would be CODE_CACHE. Otherwise, it would be ORIG_CODE, which means that it will be executed from its original location. To catch the writes to the code pages, we write-protect the code section to causes an exception. A handler is registered to catch this exception and replace the write instruction with an emulated-write, shown in figure 6.2. The emulated-write would check the destination address. If it is an original code location, it will write the value, and we will move the block to the code-cache. If the block is already in the code-cache, we invalidate the entry for that block so that the new rewritten version is copied and executed from the cache. Finally, if the address is not code, it must be data, and we will execute the write instruction normally.

For better performance, we remove write-access-protection before executing an emulated-write so that it does not trigger an exception. After the emulated-write, we will enable the write-access-protection. The number of instructions that write to code pages is limited; hence, our method's overhead is minimal. The results section will demonstrate our method's effectiveness for handling self-modifying code and

Original Write	Emulated Write

`mov dword ptr[0x0100], eax`	`Disable Write Access Protection`
	`if(d_table[0x100] == ORIG_CODE)`
	`  move the block to the code-cache`
	`else if(d_table[0x100] == CODE_CACHE)`
	`  invalidate the block`
	`else mov dword ptr[0x0100], eax`
	`Enable Write Access Protection`

Figure 6.2: Converting Write Instruction to Emulated Write Routine

writing to code pages.

6.4 Shadow Memory - Read Emulation

Obfuscated binaries might attempt to calculate and validate the checksum of code on the memory. RL-Bin sometimes replaces the original instructions with instrumentation code, and that would modify the checksum. We propose a solution that would emulate the read instructions and deceive them into reading the original value. All code pages will be read-protected so that any read from the code section causes an exception. We register a handler to catch this exception. Once a read instruction attempts reading from the code section, we replace it with emulated read, demonstrated in figure 6.3.

If an instruction is replaced by instrumentation, the corresponding value in the disassembly table is REPLACED. We maintain a map, ORIG, to keep the original bytes that are rewritten. If a read instruction reads from a modified address, we will feed it the actual value. As an optimization, emulated-read disables read access protection before reading and enable it after that, so the exception is not

Original Read	Emulated Read

`mov eax, dword ptr [0x0100]`	`Disable Read access protection`
	`if(d_table[0x100] == REPLACED)`
	`  mov eax, ORIG[0x0100]`
	`else`
	`  mov eax, ORIG[0x0100]`
	`Enable Read access protection`

Figure 6.3: Converting Read Instruction to Emulated Read Routine

triggered. Obfuscated binaries typically attempt to calculate this checksum only once, so emulated read does not have an adverse effect on the overhead.

6.5 Safe Function Detection Improvement

Obfuscation tools use multiple techniques and code artifacts that limit RL-Bin's ability to detect functions and their safety property. First, non-standard use of calls or returns means fewer functions can be detected. Next, shared blocks between functions lead to the safety property not be decided by our current method. Finally, the functions that overwrite themselves can mislead our approach to declare a function to be safe mistakenly. Once the function is overwritten, it might no longer be safe. We deem a function is no longer safe once an overwrite to the same function is detected. To handle the first two issues, we modify our algorithm to detect subfunctions. We determine the safety of these subfunctions based on how they are connected during run-time. Here are the relaxed criteria for a subfunction.

- Each subfunction has only one entry-point and one or more exit points.

- The control-flow can be redirected to the entry-point of subfunctions by any direct or indirect CTI. Also, exit-point does not need to be a return instruction and could be any indirect CTI.

- The stack pointer's value could change between the entry-point and exit-point(s) by a fixed amount.

These modifications would lead to smaller chunks of code being detected as subfunctions. During run-time, these subfunctions will be analyzed. Once the control-flow between subfunctions is determined, we declare a subfunction to be safe if the stack's return address is not modified.

6.6 Flag Register Liveness Analysis

As discussed in Chapter 5, obfuscation tools might use the flag register's value immediately after returning from a call. It would cause a problem for RL-Bin because RL-Bin's instrumentation for the return instruction overwrites the zero bit of the flag register. A high-overhead solution would be saving and restoring the flag register before and after the instrumentation routine that performs the check. Our proposed optimization is to not save and restore the flags register when possible. We would perform dynamic JIT analysis in the target of the return instruction to ensure that the value of the flag register is dead, meaning that instruction will overwrite it before being used by any other instruction. If the flag register is not live in any of the targets of return, there is no need to save and restore the flags register.

RL-Bin uses hardware and software breakpoints to follow the control-flow of

the program. The one-byte trap (0xcc in x86 assembly) for instrumentation purposes. RL-Bin inserts a one-byte trap and registers an exception handler for it. When the exception is triggered, RL-Bin will execute the instrumentation routine. As discussed in Chapter 3 , hardware breakpoints are used at the fall-through and target of conditional CTIs and triggered once the path is executed. If the obfuscated binary writes to registers which control hardware breakpoints, our proposed method is to switch to selectively using code-cache. In this case, we give up on using hardware breakpoints to handle conditional CTIs. The basic block that contains the conditional CTI will be copied to the code-cache. Unlike the original memory, we can put instrumentation in the fall-through and target in the code-cache. Once both paths are taken, we can execute that basic block from the original location. We modify the algorithm to maintain a map of one-byte traps inserted by RL-Bin++ to handle software breakpoint manipulation. Once the trap is triggered, the address will be checked against the rewriter map. If the breakpoint is not inserted by the rewriter, it would not catch the exception and not interfere with the program's intended control-flow. As a result, RL-Bin++ and the binary both will use the traps with no conflicts.

Chapter 7: Evaluation and Results

We have completed and tested a fully optimized prototype of the above method. Most of the code is written in C++, while there are some functions which are written in x86 assembly, for the sake of optimization. Our experiments are done on a system with Intel Core i7, 3.33GHz CPU with 12 Mb cache and 24.0 Gb DDR3 memory on 64-bit Windows 10 OS. We chose Windows Operating System since most commercial binaries are developed for Windows.

In our experimental setup, we used SPECrate 2017 Integer and Floating Point with their reference data sets. SPECrate Integer has 10 benchmarks demonstrated in table 7.1 and all of them are included in our testing. However, we could evaluate 10 out of 13 benchmarks in SPECrate Floating Point, which are shown in table 7.2. The other three benchmarks could not be compiled for 32-bit x86 Windows machines, thus *fotonik3d_r*, *cactuBSSN_r*, and *cam4_r* were excluded from the set.

Also, we compiled the binaries with three different compilers; Microsoft Visual Studio, GCC, and ICC. The overhead reported for each benchmark is the average of the overhead for binaries compiled with these compilers. In the case that a benchmark could not be compiled with a particular compiler, that compiler is not included in the average.

Benchmark Name	**Language**	**KLOC**	**A**pplication Area
500.perlbench_r	C	362	Perl Interpreter
502.gcc_r	C	1304	GNU C Compiler
505.mcf_r	C	3	Route Planning
520.omnetpp_r	C++	134	Discrete Event Simulation
523.xalancbmk_r	C++	520	XML to HTML Conversion
525.x264_r	C	96	Video Compression
531.deepsjeng_r	C++	10	Alpha-beta Tree Search (Chess)
541.leela_r	C++	21	Monte Carlo Tree Search (Go)
548.exchange2_r	Fortran	1	Sudoku Recursive Solution Generator
557.xz_r	C	33	General Data Compression

Table 7.1: SPECrate 2017 Integer

Comparison with DynamoRIO, Pin, and Dyninst: According to [48] and [49], we compared RL-Bin to the most efficient and robust state-of-the-art dynamic rewriters available. The reason for their robustness is the implementation using the code-cache. To the best of our knowledge, the three mentioned rewriters are the most efficient ones that are commonly used for instrumentation purposes in academia and industry.

Benchmark Name	**Language**	**KLOC**	**Application Area**
503.bwaves_r	Fortran	1	Explosion modeling
507.cactuBSSN_r	C++, C, Fortran	257	Physics: relativity
508.namd_r	C++	8	Molecular dynamics
510.parest_r	C++	427	Biomedical imaging
511.povray_r	C++, C	170	Ray tracing
519.lbm_r	C	1	Fluid dynamics
521.wrf_r	Fortran, C	991	Weather forecasting
526.blender_r	C++, C	1577	3D rendering and animation
527.cam4_r	Fortran, C	407	Atmosphere modeling
538.imagick_r	C	259	Image manipulation
544.nab_r	C	24	Molecular dynamics
549.fotonik3d_r	Fortran	14	Computational Electromagnetics
554.roms_r	Fortran	210	Regional ocean modeling

Table 7.2: SPECrate 2017 Floating Point

7.1 Run-time Overhead

The goal of RL-Bin is to perform only light instrumentation in an efficient manner. Although it can be used to perform heavy instrumentation, such as basic block counting, no binary rewriter can deliver low overhead for such instrumentation, be-

cause the added instrumentation itself is heavyweight. Hence such instrumentations are not good use-cases for RL-Bin, whose main motivation is low run-time overhead in deployed code. As a result, the run-time overhead of applications running under binary rewriter without instrumentation should be low and has been measured.

7.1.1 Overhead without Instrumentation

As it is illustrated in Figure 7.1, RL-Bin outperforms other rewriters by a significant margin, allowing it to be a more feasible choice for deployed code. In this Figure, a run-time of 100 is the run-time of the original unmodified program without rewriting. (The overhead shown as 107, means the overhead added by the rewriter is 7% without any instrumentation.) In fact, the overhead of DynamoRIO, Pin, and Dyninst for the average of SPECrate 2017 Floating Point and Integer benchmarks respectively is 1.16x, 1.26x, and 1.20 on average, whereas the overhead of RL-Bin is 1.04x for the same benchmarks (4% on average). The reason for higher overhead in Integer benchmarks is the higher number of indirect CTIs compared to Floating Point benchmarks.

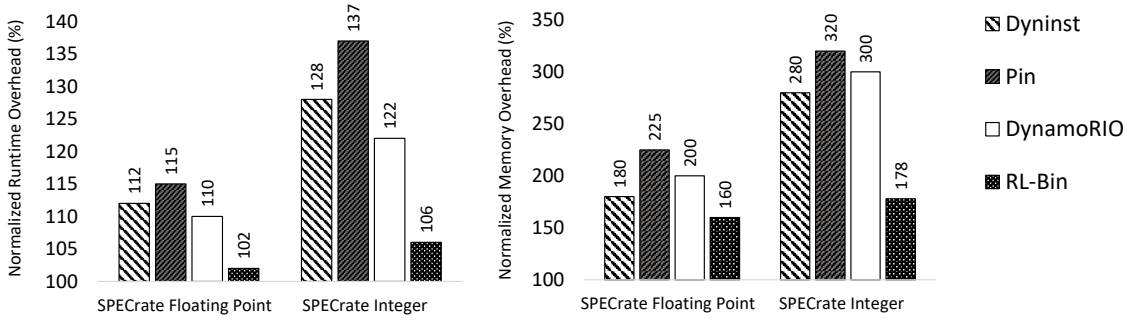


Figure 7.1: Normalized Run-Time and Memory of Rewriters Without Added Instrumentation for SPECrate 2017.

The memory overhead is also illustrated in figure 7.1. The memory overhead

of DynamoRIO, Pin, and Dyninst is 2.3x, 2.73x, and 2.5x respectively, while that of RL-Bin is only 1.69x. The main reason for lower memory overhead is that RL-Bin is an in-place design which does not rely on a code-cache, which consumes significant amount of memory.

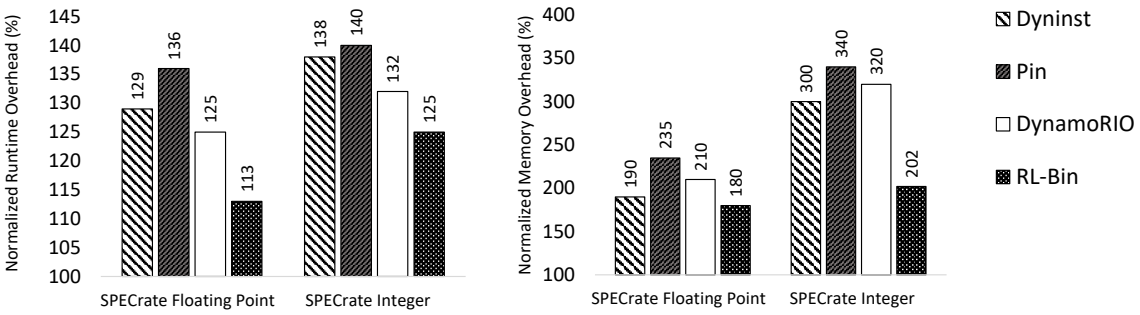


Figure 7.2: Normalized Run-Time and Memory Overhead of Rewriters with Added Instrumentations to Count External Calls for SPECrate 2017

7.1.2 Overhead with Instrumentation

The next experiment measures the run-time and memory overhead added by the rewriters when instrumenting the application to count the number of external calls from the application module to other DLLs. This particular instrumentation is used because the number of locations that need to be instrumented is relatively low. Hence, it is a good use-case of RL-Bin to perform light instrumentation with very low overhead. Figure 7.2 shows the overhead of RL-Bin with an average of 19% compared to DynamoRIO, Pin, and Dyninst which have 28%, 38%, and 33% average overhead respectively. Our experiment demonstrates that RL-Bin can be successfully used to add instrumentation with fairly low overhead compared to other dynamic rewriters.

7.2 Memory vs Run-time Trade-off and Fine-Tuning Optimization Methods

In this section, we will demonstrate how RL-Bin could be fine-tuned to decrease its memory footprint. Based on the discussions in section 4, we can fine-tune the function cloning and branch prediction optimizations to reduce memory overhead of RL-Bin.

Figure 7.3 shows the run-time and memory overheads of SPECrate 2017 benchmarks with their reference data set and no added instrumentation. This figure shows the memory and run-time of RL-Bin with different configurations for branch prediction and function cloning. In this figure, Prof means that profiling method is used for branch prediction, and Dyn1 and Dyn2 mean that dynamic branch prediction method with one or two predictions is used. By default RL-Bin is configured for best performance and clones the function at the threshold, T, and uses the dynamic prediction method with two predictions. The run-time overhead is 4% over the native execution with 69% memory overhead over the application's code segment.

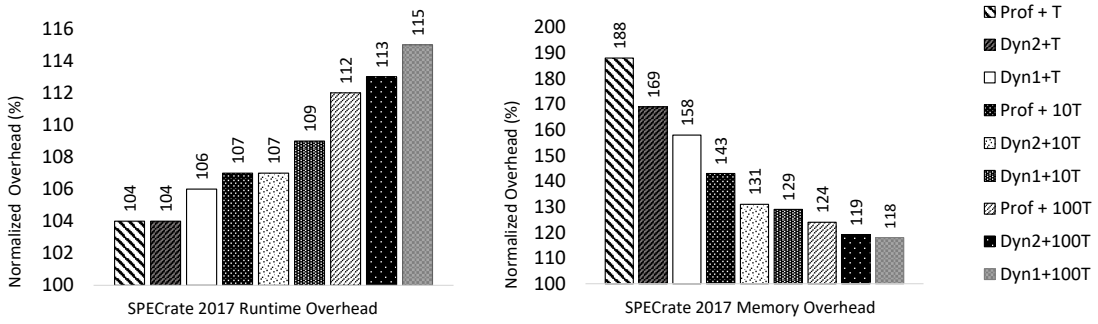


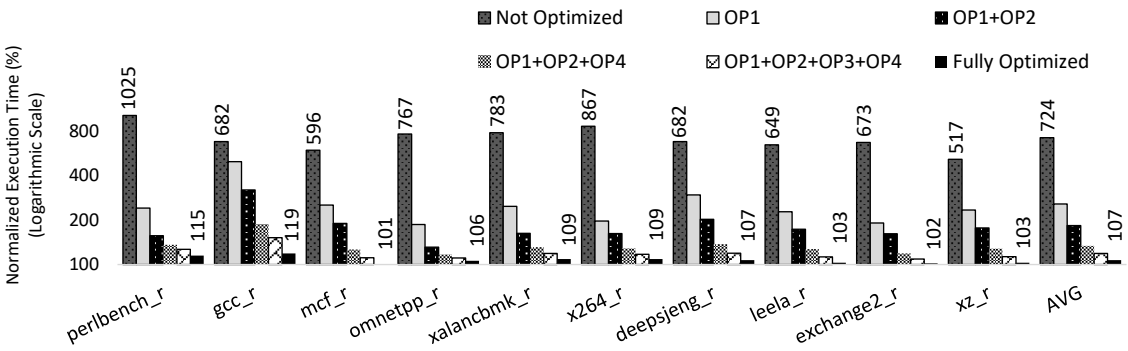
Figure 7.3: Normalized Run-Time of Rewriters Without Added Instrumentation for SPECrate 2017.

By changing the branch prediction method and function cloning threshold, it is possible to reduce the memory footprint to only 18%, at the expense of run-time overhead as high as 15%. In particular, to get satisfying memory and run-time overhead, RL-Bin can be tuned to clone functions at 10T and use the dynamic prediction with two predictions method, which would lead to 7% run-time overhead and 31% memory overhead. The correct configuration can be chosen by the user (e.g., the system administrator), based on the objectives and constraints of the system and application.

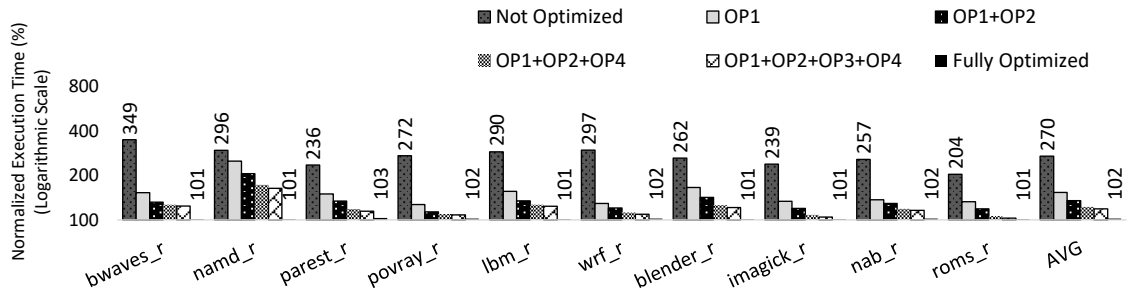
7.3 Optimization Effectiveness

To show the contribution of each optimization method proposed in section 4, we measured the overhead of SPECrate 2017 Integer and Floating Point with different optimization levels. Figure 7.4 shows the overhead with six different optimization level. For Specrate 2017 Integer, The overhead is expectedly large (10.25x for *perlbench_r*) without any optimization. Optimizing conditional branches (OP1) will bring the average overhead from 7.24x to 2.93x. Adding target prediction for indirect CTIs will reduce the overhead of remaining checks, thus the average overhead will be 2.02x with OP1+OP2. Whitelisting modules and cloning functions (OP4 and OP3) will remove lots of the added overhead for checking the target of indirect CTIs and will bring down the average overhead to 1.22x. The last set of optimizations (OP5 and OP6) detect safe functions and remove the check before the return instruction in such functions. Thus, boosting the overhead to just 1.07x on

average for SPECrate 2017 Integer benchmark suite.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 7.4: The Contribution of Optimization Methods in Reducing Overhead of RL-Bin for SPECrate 2017 Integer Without Instrumentation

7.4 Robustness

Our last experiment is designed to demonstrate that RL-Bin is robust enough to handle commercial multi-threaded applications that contain dynamically generated and self-modifying code, as well as obfuscation. We aimed to show that RL-Bin fully instruments the binary and it achieves full code coverage, meaning that no instruction is executed without being monitored by RL-Bin. The number of dynamically executed instructions was measured by instrumenting every basic block

of the application to add the size of the basic block to the total count.

7.4.1 Commercial Applications

Table 7.3 shows the list of commercial binary applications that we used in our testing. We tested three popular Microsoft Office tools; Word, PowerPoint, and Excel as well as Adobe Acrobat Reader, Adobe Premiere Pro, Adobe Photoshop, and Apache Web Server. In our experiments, in order to have dynamically generated and self-modifying code, we opened documents which contained VBA code in Microsoft Office and JavaScript in Adobe Reader. Apache Web Server heavily uses multi-threading, so this application would appropriately stress test the multi-threading capabilities of RL-Bin.

For commercial programs, we did not measure the overhead, since interaction with users and other uncertain factors, make them unacceptable as benchmarks for measuring the overhead, introduced by RL-Bin. Instead SPEC CPU 2017 was used for measuring overhead, since they are standardized benchmarks without user interaction, making them suitable for run-time measurement.

The measurements on number of dynamic instructions were done with both RL-Bin and DynamoRIO. The results showed that the numbers are the same for every application in the set, thus proving that every single instruction is counted by RL-Bin and full code coverage is achieved. As a result, proposed optimization techniques do not result in any loss of coverage, verifying that RL-Bin instrumentation is robust and accurate.

Application Name	**Feature(s)**	**Size**	**A**pplication Area
Apache HTTP Server	Multi-threading	21 Mb	Cross-platform Web Server
Microsoft Word	Multi-threading, Dynamic Code	1.8 Mb	Word Processor
Microsoft Excel	Multi-threading, Dynamic Code	57 Mb	Spreadsheet Editor
Microsoft PowerPoint	Multi-threading, Dynamic Code	1.3 Mb	Presentation Program
Adobe Acrobat Reader	Self-modifying Code	2.6 Mb	PDF Viewer
Adobe Premiere Pro	Multi-threading, Dynamic Code	2.3 Mb	Video Editor
Adobe Photoshop	Multi-threading	142 Mb	Raster Graphics Editor

Table 7.3: Commercial Applications Benchmark

7.4.2 Obfuscated Binaries

Based on the study [50], we selected three of the most popular obfuscation tools for our testing: UPX, PECompact, and ASProtect. UPX [51], short for Ultimate Packer for eXecutables, is an open-source obfuscation tool available for many platforms. It provides basic obfuscation techniques, mostly by creating indirect CTIs and writable data in the code section. PECompact [52] is a commercial obfuscation tool. In addition to the obfuscation techniques provided by UPX, PECompact generates a significant amount of self-modifying code. The third and last obfuscation tool used in our testing is a commercial tool called ASProtect [53]. It uses a

wide range of control-flow, anti-disassembly, and anti-debugging techniques and is notorious for generating binary files that are very difficult to reverse engineer.

Table 7.4 shows our feature comparison of the selected obfuscation tools. We have also included a column for original binary files for comparison. As demonstrated, the three obfuscation tools used in our testing deploy all of the challenging features discussed in the paper. Hence, they are an appropriate test to measure the robustness and overhead of RL-Bin++.

Code Artifact	**Original**	**UPX**	**PECompact**	**ASProtect**
High % of Indirect Calls/Jumps	7%	84%	44%	8%
Conditional Branch Obfuscation	✗	✗	✗	✓
Code Rewriting (opcode/argument)	✗	✗	✓	✓
Overwrite of Executing Function	✗	✗	✗	✓
Checksum of Code	✗	✗	✓	✓
Writeable Data in Code Segment	✗	✓	✓	✓
Functions Share Blocks of Code	✗	✗	✓	✓
Non-standard Calls/Returns	✗	✓	✓	✓
Flags Used Across Functions	✗	✗	✗	✓
Reusing HW Or SW Breakpoints	✗	✗	✗	✓

Table 7.4: Methods Introduced to Efficiently Handle Code Artifacts.

Table 7.5 shows the program's behavior running under RL-Bin and RL-Bin++ in the presence of the code artifacts. As described, for some cases, RL-Bin can handle

the code artifact but with high overhead. For other artifacts, the program behavior would be unexpected, leading to a crash. RL-Bin++ successfully dealt with all challenging code artifacts.

Code Artifacts	Effect on the Binary Rewriter	
	RL-Bin	RL-Bin++
High % of Indirect Calls/Jumps	High overhead (4x)	Deobfuscation reduces the overhead (1.9x)
Conditional Branch Obfuscation	High overhead (4x)	Code-cache reduces the overhead (2.5x)
Code Rewriting	High overhead (4x)	Write emulation reduces the overhead (2.4x)
Overwrite of Executing Function	Unexpected behavior	Handled by enhanced safe function detection
Checksum of Code	Unexpected behavior	Handled by Read Emulation
Writeable Data in Code Segment	High overhead (4x)	Write emulation reduces the overhead (2.4x)
Functions Share Blocks of Code	High overhead (4x)	Enhanced detection reduces the overhead (2.2x)
Non-standard Calls/Returns	High overhead (4x)	Enhanced detection reduces the overhead (2.2x)
Flags Used Across Functions	Crash/Unexpected behavior	Handled by liveness analysis
Reusing HW or SW Breakpoints	Crash/Unexpected behavior	Uses alternate methods without using breakpoints

Table 7.5: Effect of Problematic Code Artifacts on RL-Bin and RL-Bin++.

7.5 Performance and Memory Overhead for Obfuscated Binaries

In this section, we compare the overhead of RL-Bin++ with Pin and DynamoRIO. Among the dynamic binary rewriters discussed in the related works section, only Pin and DynamoRIO are mature and robust enough to properly handle the obfuscation techniques. As described at the beginning of this chapter, our experiment used SPECrate 2017 benchmarks with their reference data sets consisting of 10 Integer and 13 Floating-Point applications. We excluded three of the Floating-Point benchmarks because they could not be compiled for x86 Windows machine. We compiled the remaining benchmark programs with Microsoft Visual Studio and obfuscated them with the three obfuscation tools mentioned above.

Figure 7.5 illustrates the average overhead of RL-Bin, Pin, DynamoRIO, and RL-Bin++. As anticipated, RL-Bin only worked for original binaries and the binaries packed by UPX. RL-Bin++ can handle the binaries obfuscated by UPX, PECompact, and ASProtect. The overhead for RL-Bin++ is lower than both Pin and DynamoRIO, which are the most efficient dynamic binary rewriters based on a previous study [49]. On average, for obfuscated binaries, RL-Bin++ has 2.7x overhead while the overhead of Pin and DynamoRIO is 4.11x and 5.31x, respectively.

Finally, it is demonstrated that RL-Bin++ overhead for unobfuscated binaries is 1.06x, the same as RL-Bin, while Pin and DynamoRIO have more than 1.16x overhead. RL-Bin++ is the only robust dynamic rewriter with an overhead that is low enough for deployment in live systems.

To show the effect of each of the methods proposed in Chapter 6 , we measured

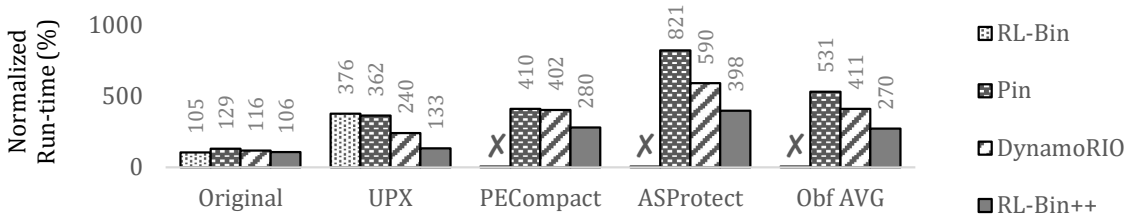


Figure 7.5: Comparison of Overhead of Original and Obfuscated Binaries Between Dynamic Rewriters

the overhead of the same benchmarks while different methods were applied successively. The result has been demonstrated in figure 7.6. It can be seen for the original binaries, the first set of columns, that the overhead remains almost the same. The reason is that none of the problematic code artifacts are present in unobfuscated binaries.

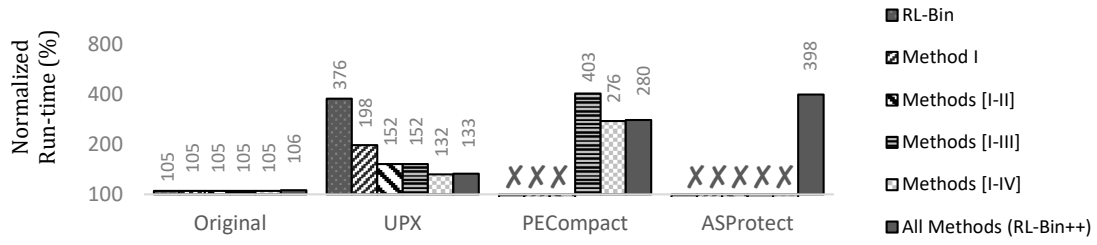


Figure 7.6: Effect of Proposed Methods on Robustness and Overhead of Obfuscated Binaries

RL-Bin can execute binaries packed by UPX with high overhead. Applying methods I, II, and III would reduce the overhead caused by a high percentage of indirect CTIs, writable data in the code section, and non-standard calls and returns. Methods IV, V, and VI do not significantly affect the overhead of these binaries.

RL-Bin cannot handle binaries obfuscated with PECompact unless methods I, II, and III are added. The reason is that self-checksum is not handled unless method III is applied. Method IV would help with detecting more safe functions and further reduces the overhead. Methods V and VI do not have a meaningful effect on the overhead.

Chapter 8: Use Cases of RL-Bin

In this chapter, we demonstrate the capabilities of RL-Bin as a base that can be used to develop complex analysis tools for practical use cases, such as to enforce security policies. In particular, using RL-Bin's instrumentations, we have designed and implemented six analysis tools, including the following; application-level file access permission tool, secure execution by restricting RETs, collect run-time properties for end-point security tool, generating guaranteed trusted disassembly, debugging and patching in deployment, and just-in-time analysis and optimization tool. First, we have developed an application-level file access permission system that enables the user to define separate access policies for each application. Second, we have created a security enforcement tool that instruments the most common form of indirect CTIs to ensure that the program execution follows the intended path. Hence, it would protect the application from being hijacked in those cases. Third, we have developed a tool which extracts run-time meta-data from dynamic execution of the application. The extracted data can be fed into a machine learning based endpoint security tool. Fourth, we have shown how RL-Bin can be used a disassembler that proviodes full code coverage without having any false positives, which is incorect disassembly of data instead of code. Fifth, we have designed a debugging in de-

ployment system which provides the unique ability to monitor the application in an external environment and fix potential issues specific to that environment. Lastly, we have shown how RL-Bin can be used as a dynamic optimization tool. In this chapter, we describe each of these use cases in detail.

8.1 Application-level File Access Permission

The primary goal of file access permission methods is to limit unauthorized users' ability to read/write/modify files containing sensitive information. The unauthorized access could be from either a malicious binary or benign program. As an example, typical ransomware attacks include encryption of the sensitive files on the system. The files cannot be accessed unless the ransom is paid to the attacker. In addition, in some scenarios, benign applications perform data extraction to gather information about the files on the system, usually for statistical analysis and data mining purposes. In the case of benign programs, the access would not harm the user, but still, it is unauthorized access performed without the user's knowledge. We are not only preventing benign applications from performing actions without the user's knowledge. We are also preventing users from knowingly or inadvertently performing actions that are unauthorized as per the organization's security policies.

To deal with the critical cases described above, the user or system administrator must have the ability to control the files that each application can access. This capability is built into the operating system like Windows and Unix. However, the security policy is enforced per file and per user, meaning that each user has per-

mission to access specific files on the system. This means that all the applications executed under file permissions of a particular user have access to the same files.

For some purposes, the OS file access permissions are not fine-grained. Consider a scenario in which the user has downloaded an application that is only supposed to access its own configuration files and never access any other file on the system. The user or system administrator should be able to determine which files can be accessed by a specific application. In the following parts of this subsection, we describe the design and implementation of an RL-Bin based system that provides fine-grained application-level permissions.

In order to develop a file access permission system, we need to have the capability of intercepting system calls. System calls are executed within the kernel and provide many services, including I/O services that access files. For the rest of this subsection, we only target Windows operating system and discuss the methods that are used to intercept system calls in Windows.

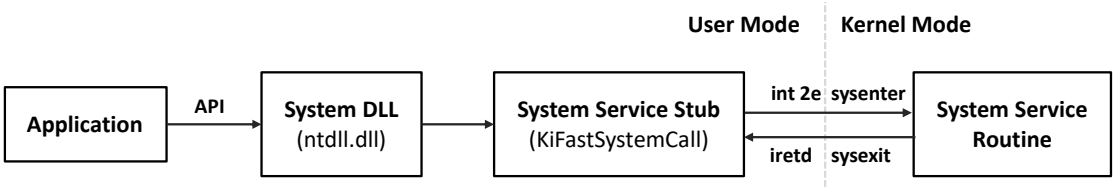


Figure 8.1: How System Calls Are Made and Possible Interception Locations

Figure 8.1 shows how applications make system calls in Windows. First, the application calls an API (Application Program Interface) function which resides in a system DLL, usually ntdll. Then the API function sets the parameters for the system call and calls the system service stub, which is the common gateway for all system calls. System service stub would use either sysenter or int 2e instruction

to transfer the control to kernel mode. System service routine in the kernel mode execute the requested system call based on the parameters that are passed and then transfers the control back to the user mode. Although this is the recommended method for accessing system calls, applications can bypass API and call the system calls directly.

There exist three methods for intercepting the system calls. First method intercept the API call. The second method intercepts the system service stub, and the last method intercept the system calls within the kernel mode. In the following paragraphs, we describe the aforementioned methods, and then propose our method for system call interception.

The first method for intercepting system calls is to intercept API function calls. In this method, the API's address on the import address table is modified to a wrapper function, which then calls the API. The wrapper function can analyze the parameters and results of the API. This method, which is usually used by patching tools, has two main disadvantages. First, using Windows APIs is only the recommended method for accessing system calls. However, it is still possible to access them directly, and this method would not intercept the system calls that are called directly. The other disadvantage is that anti-patching methods bypass the import address table and call the APIs by unconventional methods. In this case, the patching tools would miss these API; hence not all system calls will be intercepted.

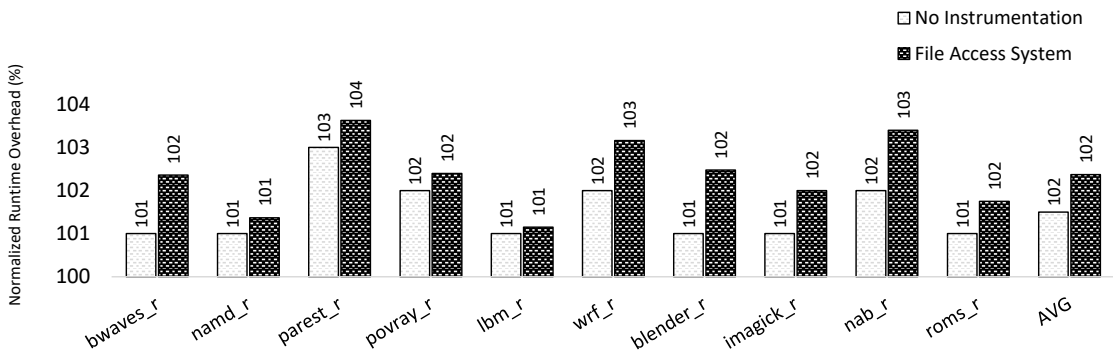
The second method is to intercept system calls at the system service stub that is shared by all system calls. As can be seen in figure 8.1, all system calls go through the same stub. This method would definitely intercept all system calls. However,

it would intercept all the system calls and not just the specific ones used for file access. The overhead of intercepting and examining all system calls would lead to high overhead, making this approach unsuitable for deployment in practice. Later on in this subsection, we show the overhead of this method for intercepting the system calls.

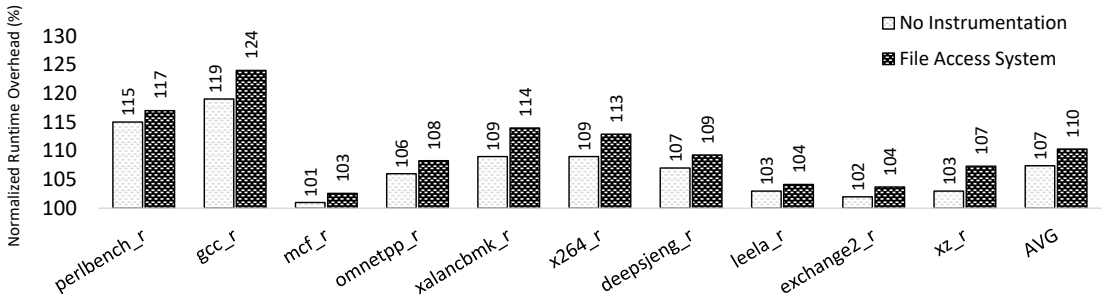
The last method that we discuss in this subsection intercepts the system calls within the kernel. The kernel code for handling different system calls needs to be changed. OS kernel modification is almost impossible for closed-source operating systems such as Windows, so this method is not suitable for implementing the application-level file access permission system.

8.1.1 Our Solution

Our approach is an extension to the first method. We instrument the binary using RL-Bin to intercept the specific system calls in which we are interested. The system calls might be invoked either directly from the application or from the body of the API function that invokes that system calls. We instrument both of these instances to intercept these system calls. By doing that, we are only intercepting the needed system calls, i.e., those used for file I/O. As a result, we avoid high overhead. In addition, we do not miss any system call because of bypassing methods used to circumvent patching tools since RL-Bin dynamically disassembles and disassembles and executes every instruction within the application and the system DLLs.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

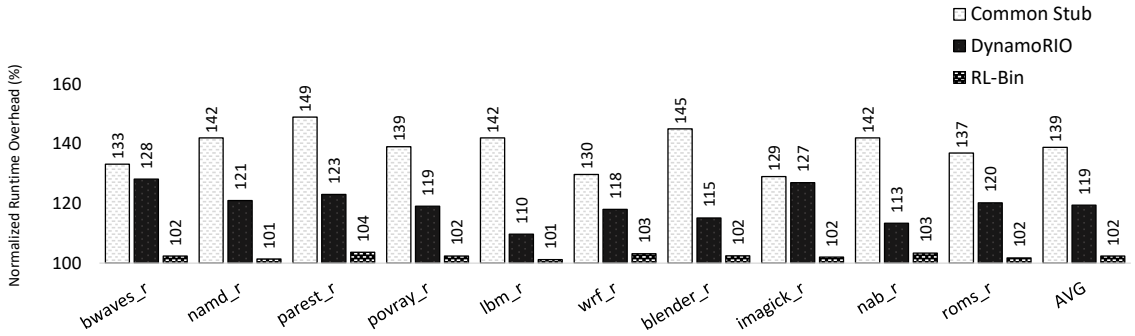
Figure 8.2: Overhead of File-Access Permission System Using RL-Bin

8.1.2 Implementation and Experimental Results

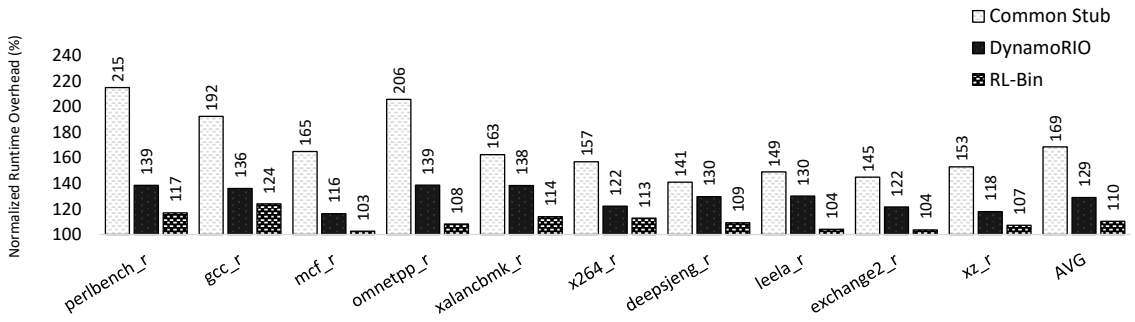
We have implemented a prototype of a file access permission system using the method described above. For this experiment, our environmental setup is the same as the results section of this paper, and RL-Bin is configured for best run-time overhead. We observed the applications in SPECrate 2017 benchmark and the files that are accessed by each application. Then we implemented a system that intercepts all the system calls and allows each application to only access the intended input and output files for that benchmark. Figure 8.2 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time

overhead of this tool based on RL-Bin is only 6% percents and it has only 2% extra overhead compared to 4% overhead for uninstrumented binaries.

For comparison, we implemented the same method by using the DynamoRIO binary rewriter. As illustrated in figure 8.3, the overhead is higher than the implementation of our proposed method using RL-Bin. As it can be seen, the overhead is 24% which is four times the overhead of implementation using RL-Bin. Additionally, we have also implemented this file access permission system by the method that intercepts all system calls at the common stub. It is shown that the average overhead of this approach is 54%, which cannot be tolerated for live deployment in end user systems.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.3: Comparison of Overhead Between RL-Bin, DynamoRIO, and Common Stub Methods

8.2 Secure Execution by Restricting RETs

A long-standing problem in security is to ensure that a benign program's control-flow is not modified. The primary method to ensure the application is not hijacked during attacks is to use a defense mechanism called Control Flow Integrity. CFI is one of the most effective application control-flow hijack defense methods invented to date and has theoretical properties ensuring its soundness and scope of the defense.

Here is how CFI works. First, the control flow graph (CFG) of the application is calculated using source code analysis, binary analysis, or execution profiling. Then CFI ensures that software execution follows one of the paths in its intended CFG. To enforce this security policy, runtime checks are instrumented before control transfer instructions to ensure that the CTI takes one of the edges from the CFG. The target address must be the destination of one of the outgoing edges from the current node. These runtime checks prevent any unintended control flow transfers during the program's execution.

CFI can protect against various attacks based on hijacking the control-flow of a benign application. These include stack-based buffer overflow attacks, heap-based jump-to-libc attacks [54], and return-oriented programming [55] (ROP). In any of these attacks, the attacker needs to transfer control to the payload code that the attacker could inject or may already be resident on the computer. During this step, CFI intercepts the CTI, checks its destination against allowed destinations, and thus terminates any attack before executing any malicious code.

8.2.1 Existing CFI-Based Tools

The Control Flow Integrity scheme was first introduced in 2005 by Abadi et al. [56]. Its goal is to monitor all CTIs to ensure that the application is following one of the edges in its CFG, which is determined in advance. Their instrumentations were added using Vulcan [57] which is a static binary rewriter. The overhead caused by added instrumentation was 16% on average for the SPEC benchmark, which is relatively high for deployment on the live systems. Some other CFI implementations have been proposed with just a few percent overhead [58]. However, these implementations require source code, and the modification would be done as part of the compiling process.

Others have tried to optimize the checks and succeeded to decrease the overhead to just about 3.6% to 8.6% [18]. However, they still rely on static binary rewriters, leading to the robustness problems we outlined in Section 1 for all such rewriters. Using a dynamic binary rewriter to perform the instrumentations for CFI has been tested [59]. The overhead was reported to be around 20%, mostly due to the high overhead caused by the binary rewriter itself. As discussed above, there is no variation of CFI implemented, which is both robust (i.e. not depending on the static analysis and inaccurate assumptions) and low overhead.

8.2.2 Our Solution

Our solution is a purely dynamic partial implementation of the CFI method (of one part of CFI for return instructions only) which does not rely on any static

information or unreliable assumptions, which which would make our solution robust and practical for all binaries. In this solution, we restrict the addresses that can be taken by return instructions and make sure returns follow calling conventions. Specifically, we use RL-Bin to insert instrumentation before return instructions to ensure that the return target address is an instruction after a call instruction.

The proposed method does not rely on a control flow graph (CFG) resulted from static analysis. Instead, we dynamically discover call and return instructions and ensure the return target is an instruction after one of the call instructions executed by that point. Although our method does not implement full CFI, it has some advantages compared to traditional CFI. The main benefit of our approach is that our approach is purely dynamic and it does not rely on any static predetermined CFG. The policy of our method provides coverage against the main category of memory-based attacks, such as return address modification and stack-based buffer-overflow attacks. It would also significantly reduce the possibility of return-oriented-programming attacks by restricting the number of potential ROP gadgets. As a future work, there exists opportunity to extend this method to restrict other types of indirect CTIs as well to improve coverage of type of attacks that this tool can prevent. In the next subsection, we would show both the effectiveness and performance of our proposed method.

One of the essential properties of a security system is that it should not cause false alarms, meaning that it should allow every non-malicious control-flow to be executed without any interruption. In our case, we must not interfere with return instructions expected to perform tasks other than returning from a function. To

avoid false positives, we detect the returns used for purposes such as stack unwinding and thread context switch. The returns that are used for stack unwinding are part of standard C++ exception handling routines. Our method detects specific sequence of instructions generated by the compiler to perform stack unwinding when handling an exception. In addition, return instructions which perform context switch, only exist in a few specific library routines within system libraries. We put all the instruction within these routines in the list of instructions that are expected not to follow usual calling convention. As described in the next subsection, our experimental results showed that all false positives could be avoided by detecting and excluding these special return instructions.

8.2.3 Implementation and Experimental Results

In this subsection, first, we demonstrate the effectiveness of the security policy on two real-world exploits. Then, we discuss the effect of added instrumentation to enforce the security policy and compare two implementations of this method by DynamoRIO and RL-Bin.

Our secure execution tool is capable of providing protection against a wide range of security attacks. According to CWE (Common Weakness Enumeration)[60], 56% of security attacks in the past year have exploited one or more vulnerability that eventually leads to a stack-based overflow attack, which is just one of the classes of attacks this method can prevent.

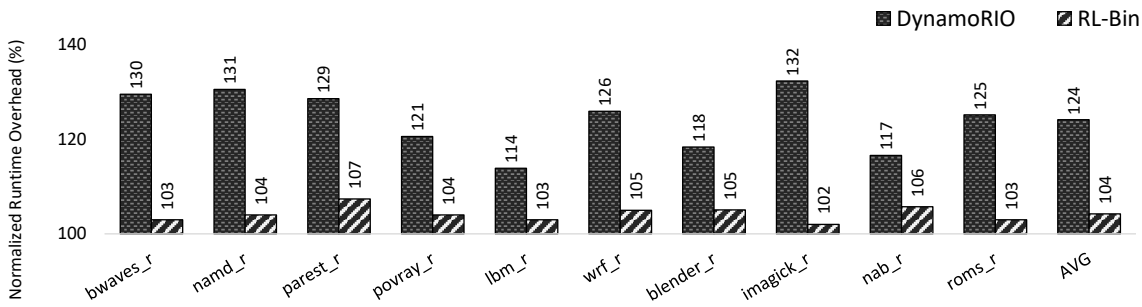
Table 8.1 demonstrates two real-world applications with the stack-based over-

flow vulnerability. We used existing proof-of-concept works to perform the attacks on these two applications. We were able to conduct the attack on the native applications successfully. Running the applications under RL-Bin with instrumentation added to ensure secure execution resulted in all intrusion attempts being detected by RL-Bin. In addition, the SPECrate 2017 benchmarks gave no false positives executed on the reference data set.

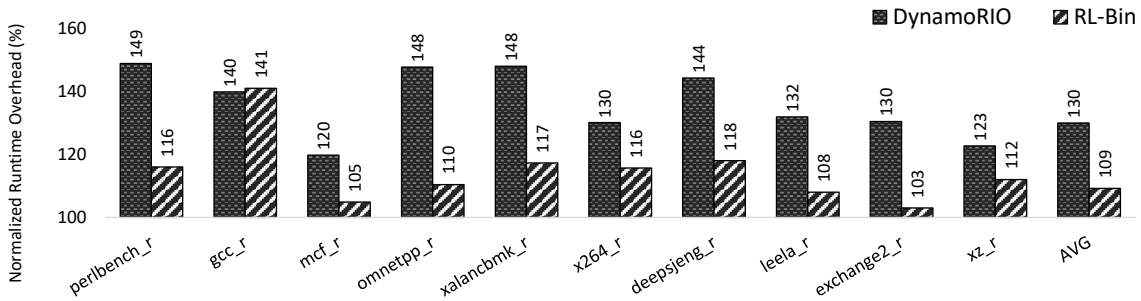
ID	Application	Description
CVE-2009-2550	Hamster Audio Player 0.3a	Stack-based buffer overflow allows remote attackers to execute arbitrary code via a playlist file.
CVE-2013-4730	PCMan's FTP Server 2.0.7	Buffer overflow allows remote attackers to execute arbitrary code via a long string in a USER command.

Table 8.1: Two Real-World Applications with Buffer-Overflow Exploits

To demonstrate the performance of our prototype, we measured the overhead of SPECrate 2017 benchmark applications using the same environment and system described in Chapter 7. Figure 8.4 shows the overhead added by instrumentation for each application compare to the native execution. As can be seen, RL-Bin's instrumentation has only 9% overhead, which makes it practical to be used in real-world systems. In comparison, the implementation based on DynamoRIO has around 27% overhead, highly unlikely to be used in practice.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.4: Comparison of Overhead of Security Policy Enforced by DynamoRIO and RL-Bin

8.3 Collect Run-time Properties for End-point Security Tool

As the program is being executed, valuable meta-data on run-time properties can be collected during execution. The meta-data that is collected include indirect branch targets, dynamic addresses accessed, exceptions taken, and the list of functions that do not return to caller. These dynamic metadata can help in just-in-time program analysis scheme (for example, a machine learning based endpoint security tool) run concurrently with RL-Bin, or in offline program analysis run after the program has executed with RL-Bin. Since this metadata information is not available prior to execution, all static tools used for program analysis do not have access to such information. Collecting dynamic information from the program enables us to

have deep knowledge and understanding about the execution of the program.

Getting dynamic metadata from the program requires modification and instrumentation of the program which needs to be done by a dynamic binary rewriter. However existing dynamic rewriters have two major problems. First, they have very high run-time overhead. Second, they are not customized for gathering these information. One may be able to use the instrumentation API provided by these tools to collect the metadata; however, most of this information is already known to the rewriter and there is no need for additional instrumentation from the user. Therefore, if existing dynamic rewriters are going to be used for this purpose, extra effort is needed from the user for adding instrumentation to the program. The added instrumentation will further increase the run-time overhead.

We collect this metadata with the lowest possible overhead, in a manner that is complete and triggered only when needed. There are two main categories of use cases of the metadata extracted from the program dynamically. First, it can be used in an endpoint security tool. Since we have access to fine-grained information from a dynamic execution of the program, we can detect program-level features from a program to detect whether it's acting maliciously or not. Another category of use cases is program analysis and optimization tools. Although several optimizations are done before compilation, there exist optimizations that can only be done during or after the execution of the program. Extracted metadata from our tool will enable optimization tools to perform such optimizations.

8.3.1 End-point Security Tool.

An example of a use case for the extracted metadata is developing an endpoint security tool. An endpoint security tool is a software tool which monitors an executing program at endpoint computer systems, and is capable of detecting malicious behavior during its execution. Existing endpoint security tools collect both static and dynamic information about the executing program. This information is distilled into features and fed to a machine learning system that predicts whether the program is malicious or not. The machine learning tool is previously trained on both known malware and benign programs, enabling it to distinguish future programs as one or the other with high confidence.

One shortcoming of existing endpoint security tools is that they only use a combination of static program header information and dynamic information about the sequence of system calls made by the executing program. Other dynamic information on the behavior of running programs, which we term "program-level features", is not collected. This information is not collected because whereas OS-level tools can intercept all system calls at low overhead, collecting program-level features requires a dynamic binary rewriter. Existing dynamic binary rewriters have high overhead, which is not tolerable in endpoint tools, since they are used in deployment. Static binary rewriters cannot disassemble code with any accuracy, and hence are not used either. Hence no existing binary rewriters are used in endpoint tools today.

An opportunity is to use metadata generated by RL-Bin to collect program-level features in an endpoint tool during the execution of the program being moni-

tored. Some program level features are typical in malicious code, and can be used as indicators fed to machine learning to increase the accuracy of malware detection. Examples of this behaviors are self-modifying and dynamically generated code, obfuscations, and packed code. Obfuscation is done by using instructions in a way that is not intended. As an example, one may register an exception handler with malicious code. Then use a divide instruction, which always divides by zero, as a control transfer instruction which redirects the flow to the malicious code. Such techniques can be easily detected by using RL-Bin. These detected features then become inputs to machine learning to improve its accuracy of malware detection.

8.3.2 Existing Tools

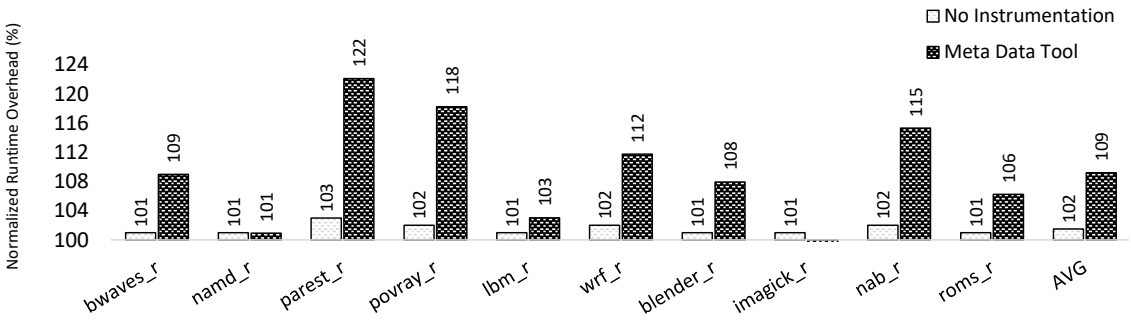
To the best of our knowledge, there is no tool that extracts run-time information, from the program. However, this would be possible by instrumenting the binary by using a dynamic instrumentation tool such as DynamoRIO[42], Pin[48], Dyninst'11 [49], Vulcan [57], or BIRD [61]. All mentioned rewriters, except for BIRD, have a high run-time overhead, more than 20% for un-instrumented binary. BIRD, on the other hand, does not support self-modification or obfuscation, so it would crash for many benign programs that have these features.

Another method would be using a debugger such as GDB [62] , OllyDbg [63], or WinDbg [64]. These debuggers are not meant to extract such runtime information; however it would be possible to log runtime metadata by tweaking the debuggers. This method would still incur high overhead and it would be impractical for use in

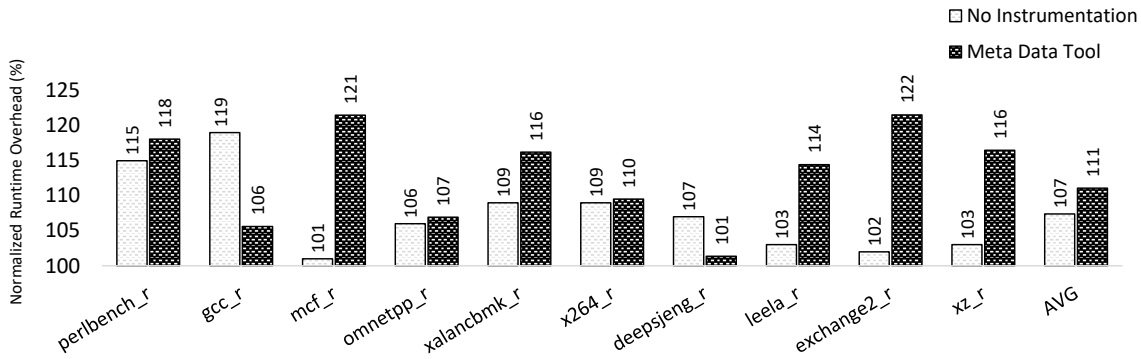
live deployment.

8.3.3 Implementation and Experimental Results

We have implemented a prototype of our tool which extracts run-time meta data. We collected the following meta data: indirect branch targets, dynamic addresses accessed, exceptions taken, and the list of functions that do not return to caller. Figure 8.5 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time overhead of the meta-data extraction tool based on RL-Bin is only 10% percents and compared to 4.5% overhead for uninstrumented binaries.



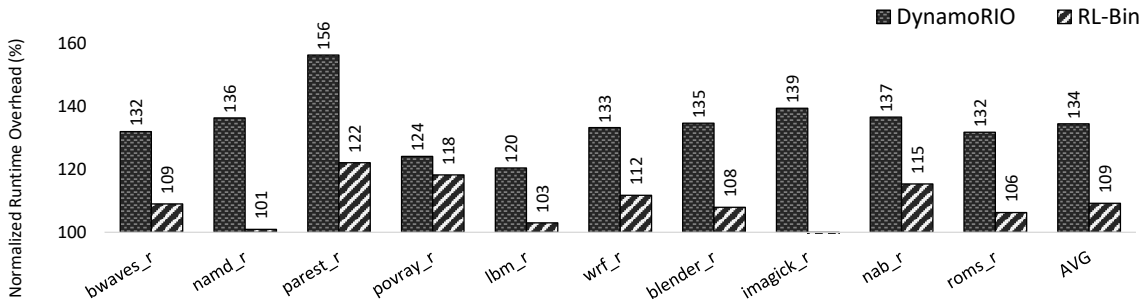
(a) SPECrate 2017 Floating Point



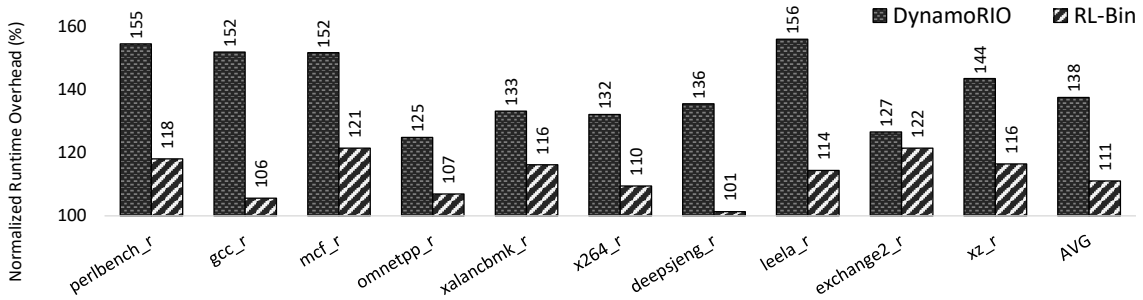
(b) SPECrate 2017 Integer

Figure 8.5: Overhead of Meta Data Extraction System Using RL-Bin

For the sake of comparison, we also implemented the same meta data extraction tool by using the DynamoRIO binary rewriter. As illustrated in figure 8.6, the overhead is higher than the implementation of our method using RL-Bin. As it is shown, the overhead is 36% which is much more than the 10% run-time overhead for the implementation using RL-Bin.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.6: Comparison of Overhead of Meta Data Extraction System Using RL-Bin and DynamoRIO

8.4 Guaranteed Trusted Disassembly

Program disassembly analyzes the program to know what the instructions in the program are, and is the basis of all program analysis tools. For many such tools, it is necessary or desirable to have a disassembly that is guaranteed to be correct,

meaning that every instruction in the output disassembly should be guaranteed to be an instruction. In general at the binary level, instructions can only be guaranteed to be so if they have been executed. Further, it is desirable to approach full code coverage, meaning that most or all the instructions should be included in the final disassembly.

Current disassembly tools are impractical. There are two main methods that is used for getting disassembly of the program; static and dynamic disassembly tools. Static disassemblers use one of the techniques mentioned earlier in section 2.2. Depending on the technique that is used, we will have either incomplete or untrusted disassembly. The main reasons are obfuscation, self-modifying or dynamically generated code, as described earlier in section 2.3. Below we describe why each of these features in code can result in incorrect disassembly.

Dynamic tools can extract the complete trusted disassembly as follows. The method is to instrument each block of the binary code so that instrumentation code extract the disassembly of that block of code. First, the entry point of the program is instrumented. Then disassembly starts at the entry point and continue until reaching a control transfer instruction. Then each of the possible target blocks are instrumented with similar code. The instrumentation in the current block is no longer needed and can be removed. The algorithm will continue in a similar pattern by executing the instrumentation code when reaching a new block of code for the first time.

However dynamic tools incur high runtime overhead, as described in Chapter 10, which makes them impractical for use in live deployment. Dynamic tools (in-

cluding RL-Bin) do not ensure complete code coverage, but approach it since every instruction that has been executed so far is included in the output disassembly, which is good enough for many just-in-time tools.

8.4.1 Existing Disassembly Tools

The most frequently used tool is objdump [65] which is a static binary disassembly tool which takes a binary as input and outputs a disassembly listing. The two main issues for objdump is that it outputs the disassembly as an unstructured text file which cannot be easily analyzed in an interactive way. The other issue is that it might disassemble data instead of code, and it is common to find *bad jumps*, jumps that are targeting data instead of code, in objdump disassembly.

Another commonly used tool is the Ida-Pro disassembler [47], which generates C-like pseudo-code whose purpose is to aid in the human understanding of the binary code. The generated pseudo-code is not meant to be executed, and often would not work if it is attempted to be compiled. Moreover Ida-Pro is not fully automatic, and may require human interaction. Further Ida-Pro's disassembly output is not guaranteed to be correct. Ida-Pro reports high disassembly coverage and the reason is that they use many speculative disassembly methods which may or may not be correct. For a program to be executed without crashing, we need a trusted disassembly, in which we can ensure with 100% certainty that every instruction in disassembly is actually code.

[15, 66] aim to achieve better code coverage by combining multiple disassembly

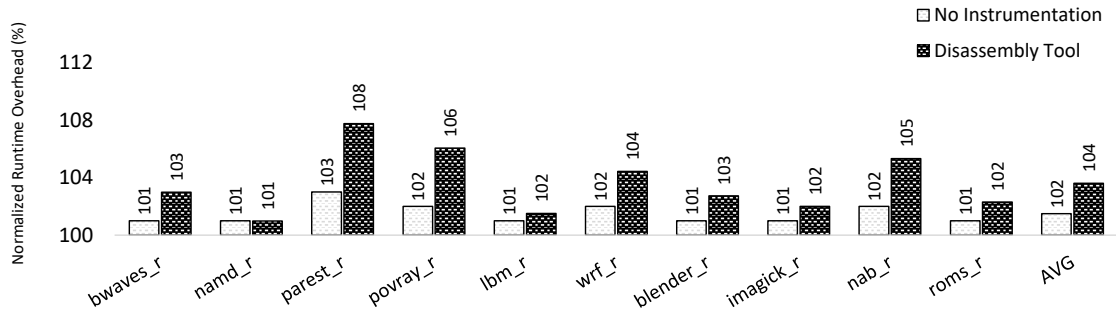
techniques including recursive traversal and linear sweep, however these tools are static and have all the shortcomings of static disassembly tools regarding dynamically generated, self-modifying and obfuscated code.

Dynamic debugging tools such as gdb [62] could also be used for disassembly. The gdb tool uses symbolic information or source code if provided, otherwise it shows the disassembly of the instructions that are currently executing. This would ensure correct disassembly since the instructions that are shown are either being executed or have been executed before. The main disadvantage of this tool is that disassembly cannot be stored in a way which is suitable for later analysis and also this method has a very high runtime overhead.

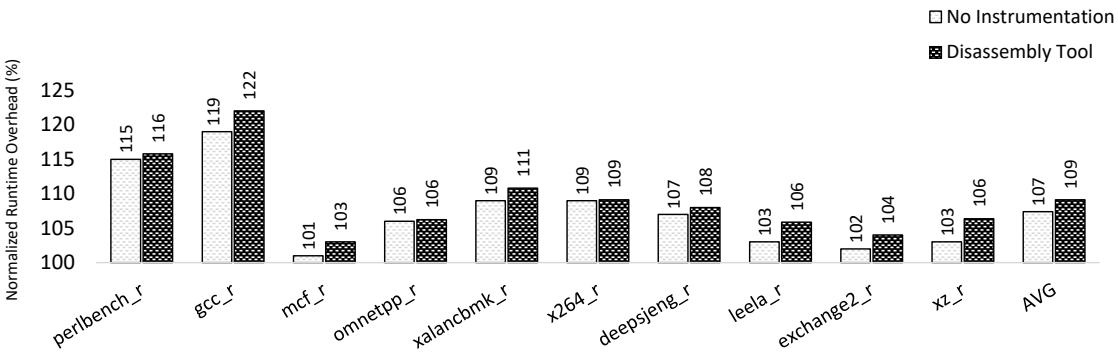
Another tool which aims to disassemble a program is BIRD [61], a low-overhead tool which uses a hybrid of both static and dynamic disassembly. Initially, BIRD disassembles the program using speculative static disassembly. Each memory location has a confidence score which shows the probability of that memory location to be code. Later on, while the program is being executed, the confidence score is updated. In addition if a new memory location is discovered as code, it would be disassembled. If confidence score is more than a certain threshold, then that memory location is marked as code in the final disassembly. This method does not ensure 100% trusted disassembly; however; in most cases, disassembly is accurate, Although the runtime overhead of BIRD is acceptable, the problem is the inability to disassemble self-modifying code, no support for obfuscation, and the lack of 100% correct disassembly.

MULTIVERSE [67] tries to achieve better code coverage by Superset Disass-

mebly. Their approach disassembles the binary code into a superset of instructions, and also benefits from MULTIVERSE, an static binary rewriter that is capable of relocating instructions by redirecting all indirect control flow transfers.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.7: Overhead of Disassembly Tool Using RL-Bin

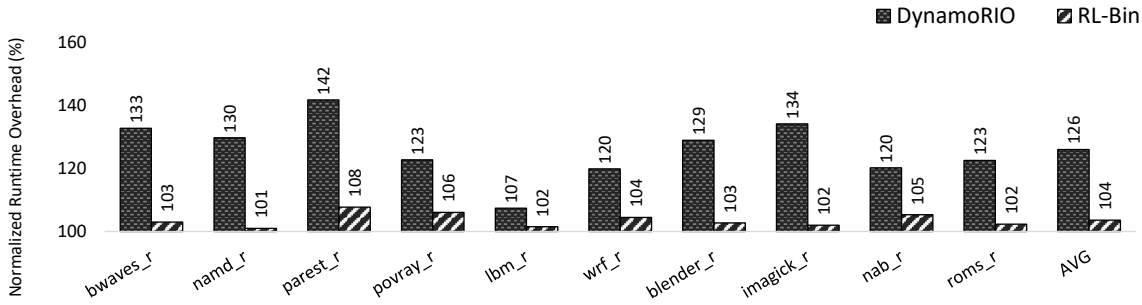
8.4.2 Implementation and Experimental Results

We have modified RL-Bin to output disassembly of the program as it is being executed, in a manner that is correct and low overhead. The disassembly provided by RL-Bin contains every instruction that is executed. This guarantee flows from the property of RL-Bin that it monitors all the code as it executes, and never loses control of the program. Moreover no data is mistakenly output as code, since RL-Bin

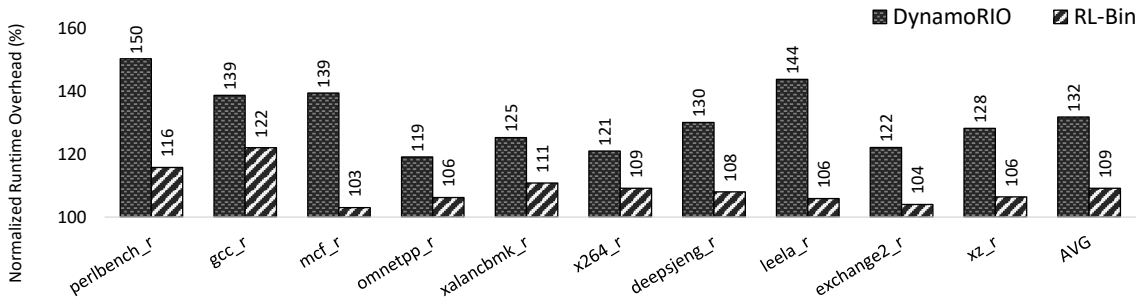
only outputs instructions after they are executed.

We have implemented and tested a prototype of our disassembly tool. Figure 8.7 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time overhead of the disassembly tool based on RL-Bin is only 6.5% percents and compared to 4.5% overhead for uninstrumented binaries.

Again for comparison, we also implemented the disassembly tool by using the DynamoRIO binary rewriter. As it is shown in figure 8.8, the overhead is higher than the implementation of our method using RL-Bin. As it is shown, the overhead is 29% which is much more than the 6.5% run-time overhead for the implementation using RL-Bin.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.8: Comparison of Overhead of Disassembly Tool Using RL-Bin and DynamoRIO

8.5 Debugging and Patching in Deployment

Making sure that the application is running flawlessly is one of the most arduous tasks in software development process. In practice, it is often the case that programs face run-time errors, or show unexpected behavior. The main reason is insufficient test data sets for different scenarios. End user systems will have different resources, and configuration. An error may arise only in certain execution platforms, and never come up in development tests. As a result, debugging is needed even after the development process.

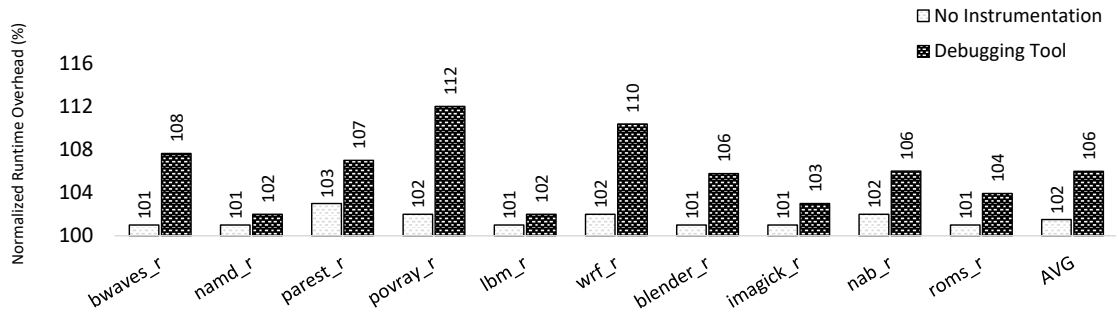
Now, consider the following scenario. The developer has released the software to the end user, but there is a bug in the software which only happens in the end user system. The developer cannot reproduce the error in the development environment. There are two existing methods to solve this problem. First, the program may be executed with the presence of a debugger to find where the issue happens. However, almost all commercial binaries are stripped of their debug information to protect their code from being reverse-engineered. As a result, this solution is impractical and the developer will not share debugging information with the user. Another solution is to generate an error log whenever the application crashes and send it to the developer. Log file may contain current stack, and the value of certain attributes of the program. This may be useful to learn more about the issue, however, it is too general, the developer will need extra information. In addition, neither of the methods above would patch the code and solve the issue. Even if the error is found, the user needs to wait for the next release of the application which may take a long

time. If the bug is a security concern, it is crucial to patch the program as soon as possible.

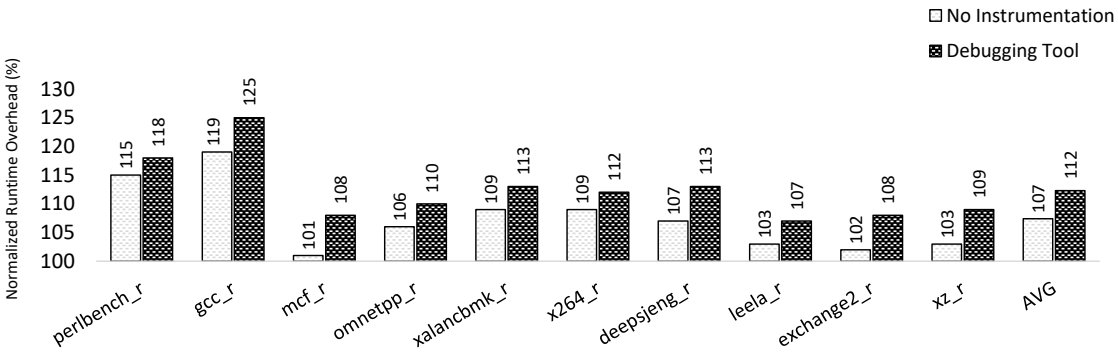
There exist tools in the literature [68] that would propose automatic approaches for pin-pointing the causes of the software failures. RL-Bin can also be used as the base tool for instrumenting those application without high overhead.

8.5.1 Our Solution

Our solution takes as input debugging information of the program and any arbitrary instrumentation that the developer wants to put in the program. We recompile RL-Bin to use the information of the debug file and generate instrumentation that will be inserted in the target application. Based on the debug file, RL-Bin would know where to instrument. The modified version of RL-Bin, dynamic debugger, will be sent to the end user. Added instrumentation will monitor execution and send requested information to the developer. Thus, enabling the developer to pinpoint the problem and fix the issue. This dynamic debugger does not reveal debugging information to the end user. Only recompiled RL-Bin is sent to the end-user system and the debug information file never gets exposed. Another advantage is that the code can be patched dynamically when the binary is being executed. This is crucial for certain service applications which need to be responsive all the time.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.9: Overhead of Debugging in Deployment System Using RL-Bin

8.5.2 Implementation and Experimental Results

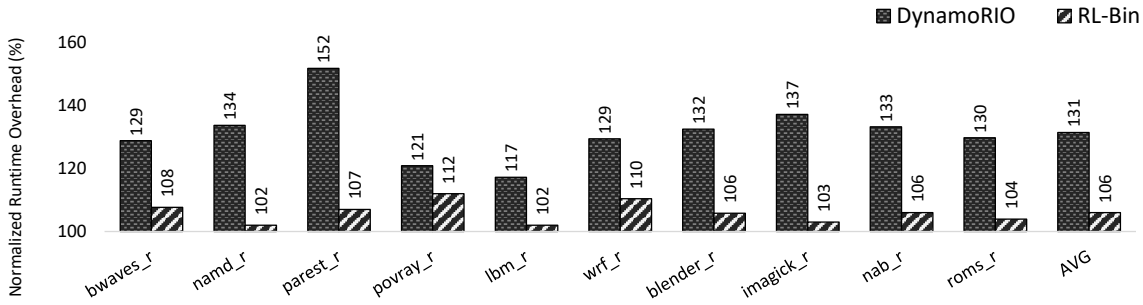
As a proof of concept, we developed a prototype of our dynamic debugger. This prototype is capable of parsing PDB file format which stores debugging information of the programs compiled with Microsoft Visual Studio. Our debugger will instrument the program to monitor it during its execution.

We implemented and tested a simple use case. Ten random functions are chosen in each of the applications in SPEC CPU2017 benchmark, and then instrumentation is added to report the maximum value of the first argument passed to each of these ten functions during the execution of the program. (The purpose of

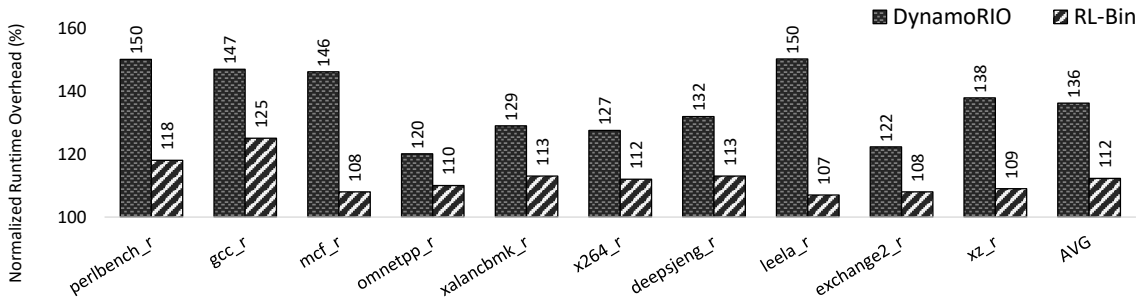
our test was to measure the overhead, the actual functions and the way that it needs to be monitored depends on the developer and may vary case by case.)

Figure 8.9 shows that the average overhead is just 9%, which means that added instrumentation to monitor these functions added 4.5% extra overhead in comparison to 4.5% overhead for binaries without added instrumentation.

For the sake of comparison, we also implemented the debugging tool by using the DynamoRIO binary rewriter. As illustrated in figure 8.10, the overhead is higher than the implementation of our method using RL-Bin. As it is shown, the overhead is 33% which is much more than the 9% run-time overhead for the implementation using RL-Bin.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.10: Comparison of Overhead of Debugging in Deployment System Using RL-Bin and DynamoRIO

8.6 Just-in-time Analysis and Optimization Tool

Program performance can be boosted by performing optimizations that are customized based on the input values provided to the program, and also the properties, resources and configuration of the system. The optimizations that can be done dynamically have three main objectives: (i) adaptive optimization of the program based on the input values provided to the program; (ii) optimizing the code in order to effectively use the resources of the system; and (iii) performing program optimizations that require run-time information, such as indirect branch optimization for the common case, and program level inter-function optimizations. We will further investigate one or more categories of these optimizations in order to show the capabilities of our tool. We discuss each of the three categories in turn below.

First, program execution is very dependent on the input values provided. For example, a run-time variable might be the number of times that a loop is executed. Depending on that value, some loop optimization techniques may be done. Generally, this type of optimization is done by using the information extracted from profiling. However profiling has two drawbacks: (i) it places the burden on the user of the program to collect profiles and re-optimize the program; and (ii) profiling encapsulates average-case behavior when run on a representative input, but does not re-optimize each time a different input is provided. A run-time program optimization system at the binary level based on RL-Bin can automate optimization during the run of the program itself without user involvement, and optimizes for the input in each individual run and data inputs, rather than a single profile input.

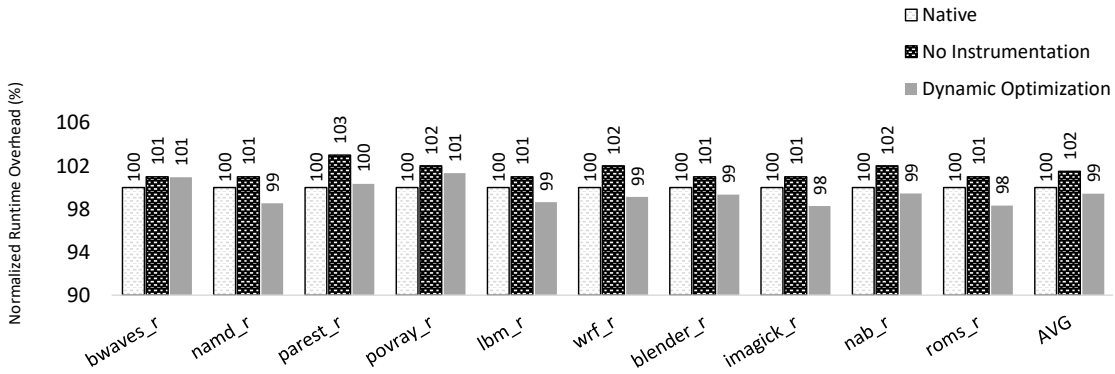
Second, resource-based optimizations can be done that significantly boost the performance of the system. Programs are compiled and optimized for specific architectures. However, they're not optimized for every end-point system. Just-in-times optimizations can be done based on the resources of the system such as number of processor cores, amount of memory, and network connection speed. As an example, we can perform optimizations such as loop unrolling and function cloning which boost performance of the program at the expense of extra memory overhead. On the other hand, we may want to perform code de-bloating optimization if the system does not have sufficient memory.

Finally, we gain the ability to optimize parts of the code using run-time values. One simple example is that an indirect branch or indirect call may actually point to one location most of the time at run time. This indirect branch can be modified and changed to a direct branch with an extra check to make sure that the assumption about the branch is correct. Such optimizations can help boost the performance of the system, because of the fact that indirect CTIs cause significant overhead since they cannot be predicted and statically optimized.

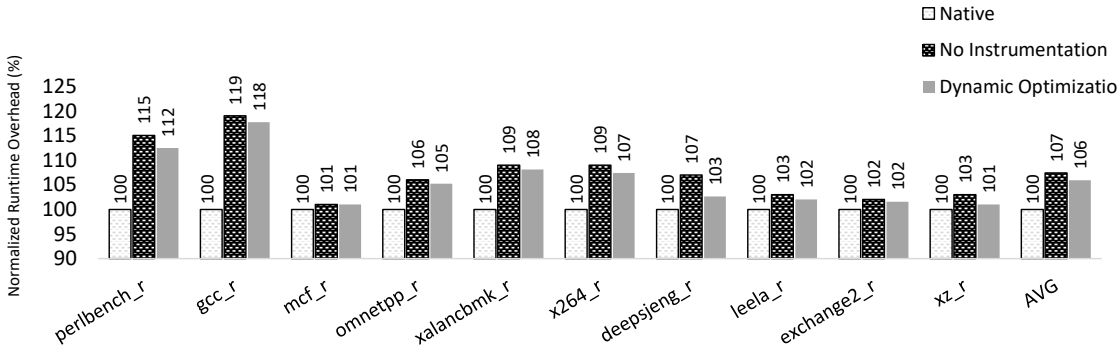
8.6.1 Implementation and Experimental Results

As a proof of concept, we implemented the last optimization described in previous subsection. We used RL-Bin to convert an indirect branch or indirect call to a direct branch with an extra check to make sure that the assumption about the branch is correct. Our prototype will convert those indirect CTIs which have only

one destination to a direct CTI. Figure 8.11 shows that this dynamic optimization can reduce the overhead of Specrate Floating Point applications by one percent. However, the overhead of RL-Bin for SPECrate 2017 Integer benchmarks is high and the benefit of optimization cannot be gained for these benchmarks.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 8.11: Overhead reduction of Dynamic Optimization Tool Using RL-Bin

Chapter 9: Application User Interface

9.1 Customizable Easy to Use Interface for Instrumentation

Having a customizable and easy to use API is one of the most important features of a binary rewriter. Existing rewriters such as Dyninst, PIN and DynamoRIO have expressive and flexible APIs [69, 70, 71] that enable users to specify instrumentations at the function, basic block, or instruction level. Defining a general-purpose, complete, efficient, and flexible set of APIs avoids the need for the user of the binary rewriter to understand and modify its source code, a process that is time consuming and error prone. The better the APIs are, the less effort is needed from the user to instrument a program for a specific use case.

Our custom instrumentation API has functionalities similar to existing APIs. However, the implementation is different and the main reason is that our rewriter is not based on a code cache. As a result, unlike DynamoRIO and PIN, we do not focus on trace-level instrumentation. The focus of our API is efficiency. In our initial prototype, we provide APIs for the functionalities listed below.

- Program control and initialization
- Analyzing and instrumenting a routine/function/procedure within a section

- Analyzing and instrumenting a basic block or instructions within the basic block
- Analyzing and modifying the arguments passed to routines
- Providing thread and process support
- Providing system call support
- Accessing debug information
- Analyzing and modifying hardware and software exceptions

We have designed and implemented an instrumentation API for RL-Bin that is both efficient and flexible. We have similar set of APIs compared to existing tools, so that users who are familiar with currently available tools can adapt to RL-Bin with minimal effort.

Custom Instrumentation APIs in Existing Rewriters

Most advanced binary instrumentation tool enjoy the benefits of providing a custom instrumentation API to the user. Examples of these APIs include DynamoRIO's API [70], Pin's API [71], and Dyninst's API [69]. All of these rewriters provide a robust and flexible API which can be used for instrumentation in instruction, basic block, trace, or function level. Pin's API has some methods to instrument child processes, multiple threads, and monitoring operating system interaction with the program. DynamoRIO's API can steal a register from the program and use it only for instrumentation. It also provides some optimization techniques for reducing

overhead of persisting instrumentation code. We will borrow the best ideas from these tools in designing the API for RL-Bin. Since RL-Bin's internal structure is not similar to the tools above, the implementation of the APIs is likely to be different.

9.2 Example Instrumentation Using API

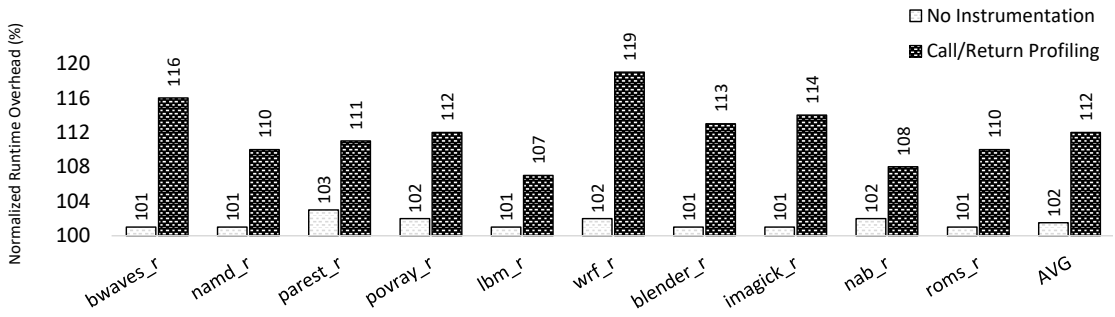
This section gives three example usecases of application programming interface of RL-Bin: Call/Return Profiling using our function profiling interface, Basic Block Counter using our basic block interface, and Conditional CTI Profiling using our instruction profiling interface. In the following subsections, we demonstrate the result of instrumentation performed by RL-Bin and compare it with the same instrumentation using DynamoRIO.

9.2.1 Call/Return Profiling

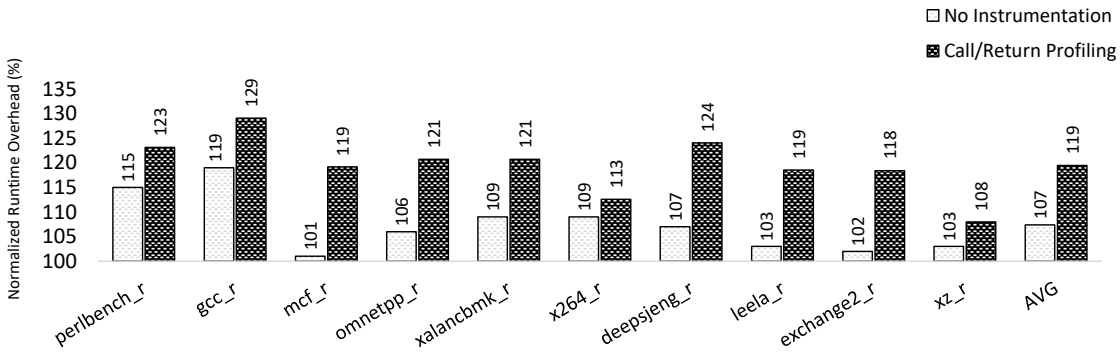
This experiment measures the run-time overhead added by RL-Bin when instrumenting the application to count the number of direct and indirect calls and return instructions through the dynamic execution. In this particular instrumentation, the number of locations that need to be instrumented is comparatively low.

Figure 9.1 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time overhead of Call/Return Profiling instrumentation based on RL-Bin is only 15.5% percents and compared to 4.5% overhead for uninstrumented binaries.

Figure 9.2 shows the overhead of RL-Bin with an average of 15.5% compared



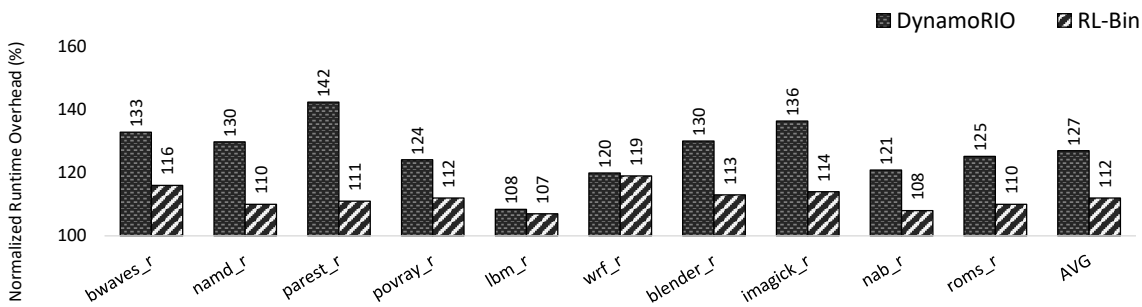
(a) SPECrate 2017 Floating Point



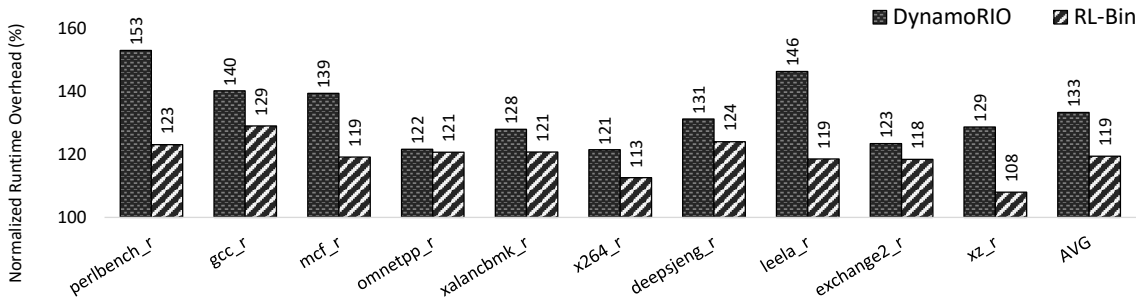
(b) SPECrate 2017 Integer

Figure 9.1: Overhead of Call/Return Profiling Using RL-Bin

to DynamoRIO which has 30% average overhead for the similar instrumentation. Our experiment demonstrates that RL-Bin can be successfully used to add instrumentation with fairly low overhead compared to DynamoRIO.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

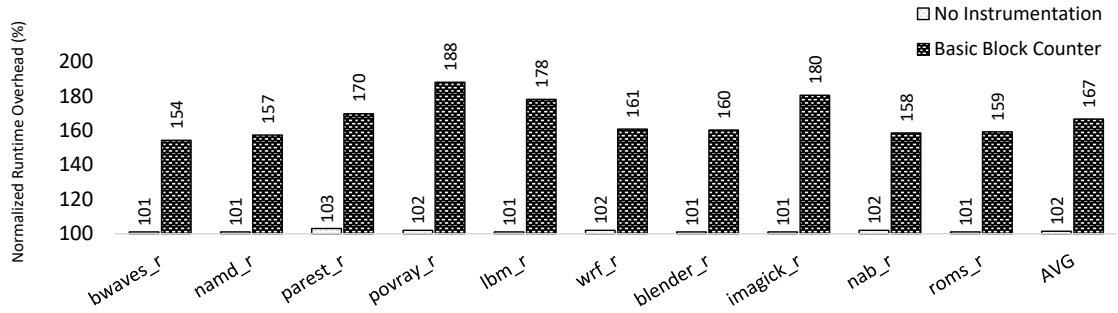
Figure 9.2: Comparison of Overhead of Call/Return Profiling Using RL-Bin and DynamoRIO

9.2.2 Basic Block Counter

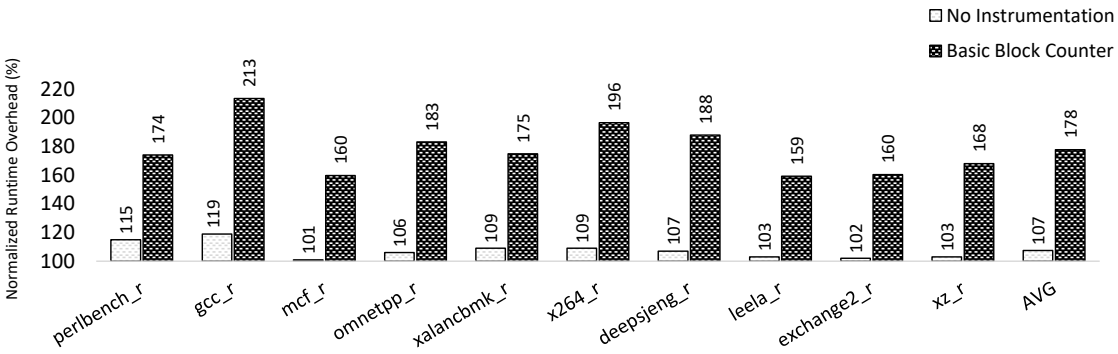
This experiment measures the run-time overhead added by RL-Bin when instrumenting the application to count the number of basic blocks that were executed dynamically. For this instrumentation, the number of locations that need to be instrumented is very high. RL-Bin is not designed for these type of instrumentations.

Figure 9.3 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time overhead of Basic Block Counter instrumentation based on RL-Bin is 72.5% percents and compared to 4.5% overhead for uninstrumented binaries.

Figure 9.4 shows the overhead of RL-Bin with an average of 72.5% compared



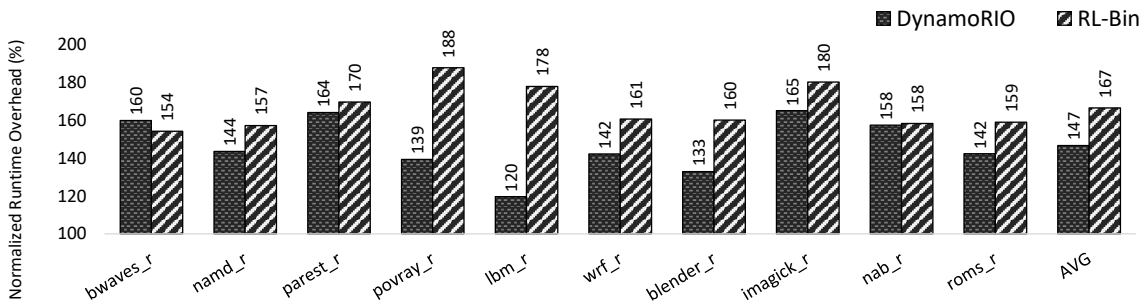
(a) SPECrate 2017 Floating Point



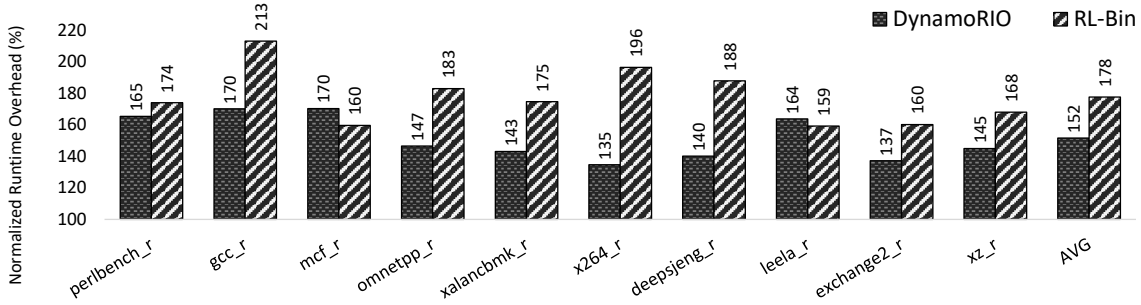
(b) SPECrate 2017 Integer

Figure 9.3: Overhead of Basic Block Counter Using RL-Bin

to DynamoRIO which has 49.5% average overhead for the similar instrumentation. Our experiment demonstrates that RL-Bin does not have the performance of DynamoRIO when the amount of instrumentation is very heavy. The main reason is that RL-Bin uses an in-place design instead of a code cache. As a matter of fact, RL-Bin is designed for light instrumentation and the goal is to be deployed in live system. No existing rewriter, including RL-Bin, DynamoRIO, Pin, and Dyninst, can perform heavy instrumentation with an acceptable overhead for live deployment.



(a) SPECrate 2017 Floating Point



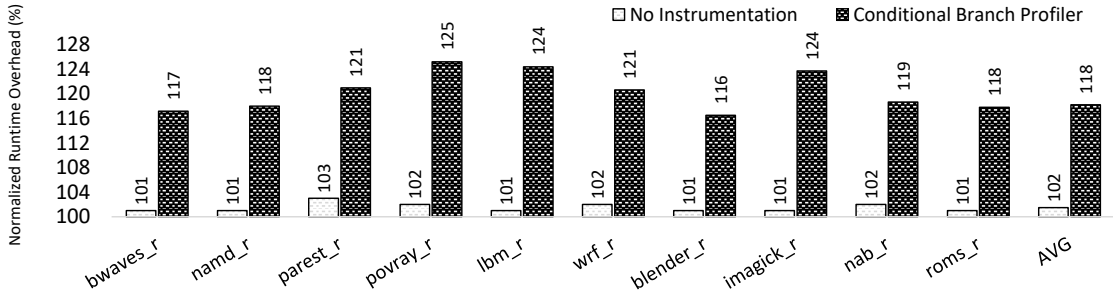
(b) SPECrate 2017 Integer

Figure 9.4: Comparison of Overhead of Basic Block Counter Using RL-Bin and DynamoRIO

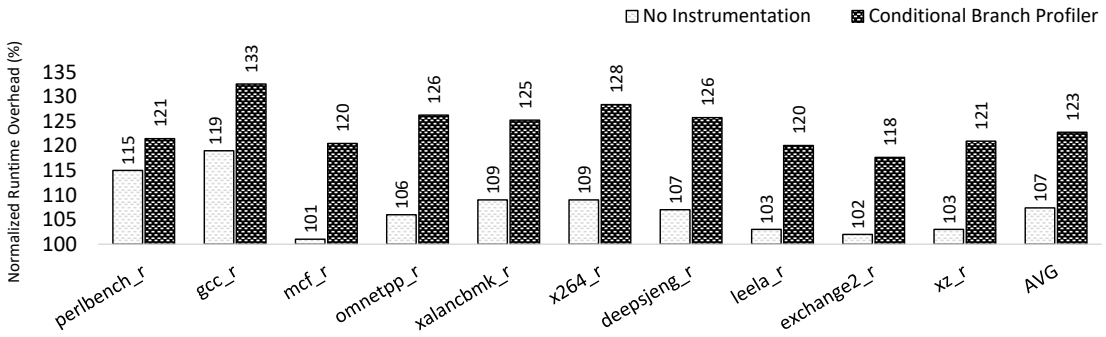
9.2.3 Conditional CTI Profiling

This experiment measures the run-time overhead added by RL-Bin when instrumenting the application to count the number of taken and not taken conditional branches during dynamic execution. For this instrumentation, the number of locations that need to be instrumented is fairly high, which RL-Bin is not designed handle very well.

Figure 9.5 shows the overhead of each application in the SPECrate 2017 benchmark suite. The average run-time overhead of Conditional CTI Profiling instrumentation based on RL-Bin is 20.5% percents and compared to 4.5% overhead for uninstrumented binaries.



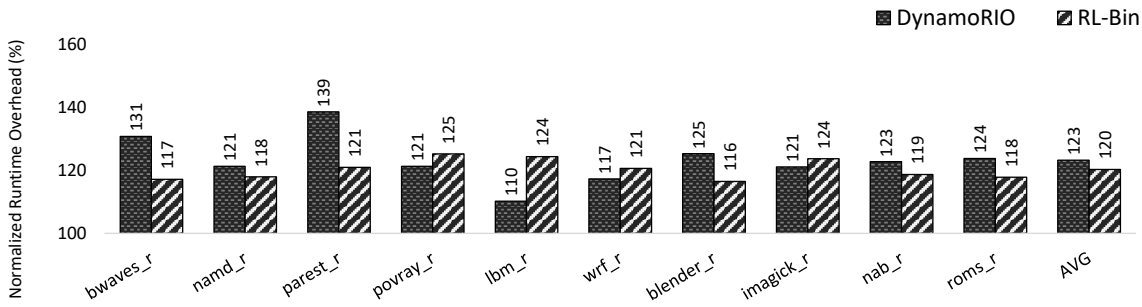
(a) SPECrate 2017 Floating Point



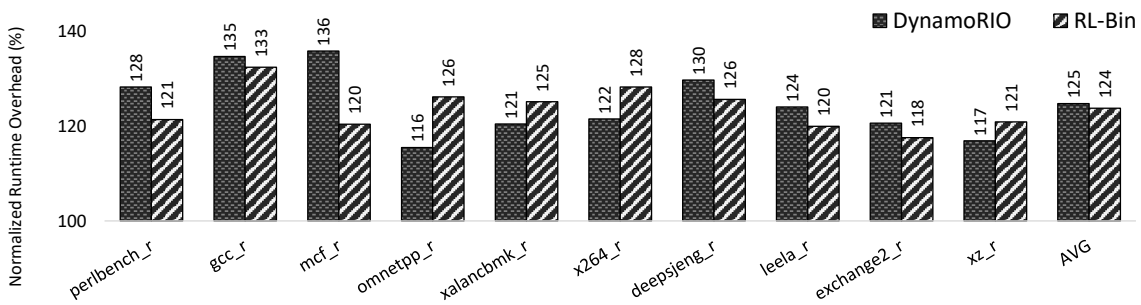
(b) SPECrate 2017 Integer

Figure 9.5: Overhead of Conditional CTI Profiling Using RL-Bin

Figure 9.6 shows the overhead of RL-Bin with an average of 20.5% compared to DynamoRIO which has 24% average overhead for the similar instrumentation. This experiment shows that although RL-Bin is not designed to perform heavy instrumentation, its overhead is still very competetive to other dynamic rewriters such as DynamoRIO. Still, this type of instrumentation are not practical for deployment in any live system. With this experiment, we demonstrated that the performance of RL-Bin for off-line use cases is nearly as good as other dynamic instrumentation frameworks.



(a) SPECrate 2017 Floating Point



(b) SPECrate 2017 Integer

Figure 9.6: Comparison of Conditional CTI Profiling Using RL-Bin and DynamoRIO

Chapter 10: Related Works

Binary rewriting is a well researched field of study and during the past thirty years, there has been several major rewriters developed to address specific needs of the community. [72] thoroughly covers existing works in full depth. Figure 10.1 shows the advantages and disadvantages to static rewriters and other dynamic rewriters. In this chapter, we will dicuss static and dynamic rewriters in details and compare them with RL-Bin.

Static
Advantages
- Low-overhead
Limitations
Does not work for: - Self modifying code - Obfuscation - Binary file modified on the disk

RL-Bin
Advantages
- Low-overhead for light instrumentation - Can rewrite every program
Limitations
- High overhead for heavy instrumentation

Dynamic
Advantages
- Rewrite every program - Reasonable overhead for heavy instrumentation
Limitations
- High-overhead, so can be used for testing, but impractical for use in deployment.

Figure 10.1: Comparison of Advantages and Disadvantages of RL-Bin with Static and Dynamic Rewriters

10.1 Static Binary Rewriters

Currently, lots of static rewriting solutions are available including [14, 73, 74, 75, 76, 77, 78]. SecondWrite project [73] aims to recover compilable source code from binaries, initially output as LLVM IR, which could then further be compiled into rewritten executable code. ATOM [74] and Vulcan [57] provide flexible interfaces for code instrumentation which help in the development of program analysis tools. Dyninst's 2007 version [49, 79] is an in-place static binary rewriter aiming to provide low-overhead instrumentation capability. Pebil [75] is another static binary rewriter focused on achieving efficient binary instrumentation by using function-level code relocation for inserting control structures.

Another type of static binary rewriters include Etch [11] , Squeeze and Squeeze++ [80, 81] , OM [82] , ALTO [83], PLTO [14], Spike [84, 85] and Diablo [86, 87]. These are optimizing binary tools or object-code rewriters. These static rewriters are not particularly used for general instrumentation purposes and are mostly used in optimizations. The input of these static rewriters are object files and not the binary. Hence these tools can only be used by software developers. As an example, Diablo [86] aims to provide a framework for link-time program transformation with whole program optimization. Unlike these static rewriters, RL-Bin is not dependant on relocation infromation that is stripped from most of commercial binaries.

Static rewriters, including all of the above, face significant limitations due to the lack of run-time information when trying to disassemble and instrument the binary. The first limitation is that they cannot disassemble dynamically generated

or self-modifying code. The reason is that these codes are not available before the execution of the program. This will lead to incomplete code coverage.

Dynamically generated code is quite common in benign applications. In a recent study [88], it was observed that 29 out of 120 benign applications contain dynamically generated code, which is used for supporting execution of user scripts. This means that implementations of security policies which use static binary rewriters would fail for 24% of applications.

The second limitation of static binary rewriting arises from the fact that some benign programs contain data in their code segment. Static disassemblers aim to understand the contents of code segments using two types of disassembly – linear sweep or recursive traversal. Linear sweep ensures high code coverage. However, it cannot distinguish between real code and data in the code segment.

To overcome the problem of data in code segments, another method of disassembly must be used. This method is recursive traversal, which only treats a region of the code segment as code if it can statically prove a control-flow path to it exists. static control flow paths are only known through direct CTIs. For indirect CTIs, the targets are not statically known and the target is only reachable via indirect CTIs.

A third limitation of static binary rewriting is that some benign programs contain obfuscated code, in which case static rewriting can break the program. The relevant kind of obfuscation is control-flow obfuscation whose goal is to mislead disassemblers so that they cannot reverse engineer binaries.

10.2 Dynamic Binary Rewriters

There are two main types of dynamic binary rewriters: in-place designs, and code-cache based designs. We will go over them briefly.

In-place designs, such as BIRD [61] have lower over-head in comparison to code-cache based designs by avoiding the high overhead incurred by maintaining the code cache; however, they fail to support some of the features which may happen relatively frequently in benign binaries such as obfuscation, dynamically-generated and self-modifying code. The reason BIRD does not work for obfuscated code is that it assumes both the fall through and destination of a conditional branch are code, which may not be true in obfuscated code. Further, BIRD does not support self-modifying code. The reason is that once they disassemble code from a location, they never change the disassembly even if the code is overwritten.

Unlike static and in-place dynamic rewriters, **code-cache based** dynamic rewriters are robust and can correctly rewrite all programs. However existing rewriters have high overhead that is generally unacceptable for deployment on live systems. Two of the most popular code-cache based dynamic rewriters are DynamoRIO [42, 89] and Pin [48, 90] with 1.2x and 1.54x run-time overhead, respectively, on average for the full SPEC'06 benchmark suite *even without any instrumentation inserted.* Dyninst'11 [91] is another code-cache based design which has 1.2x overhead for the same benchmark. Valgrind [92] is another dynamic binary rewriter which has a very strong API for adding instrumentation; however, it has very high overhead, around 2x to 3x compared to un-instrumented binaries.

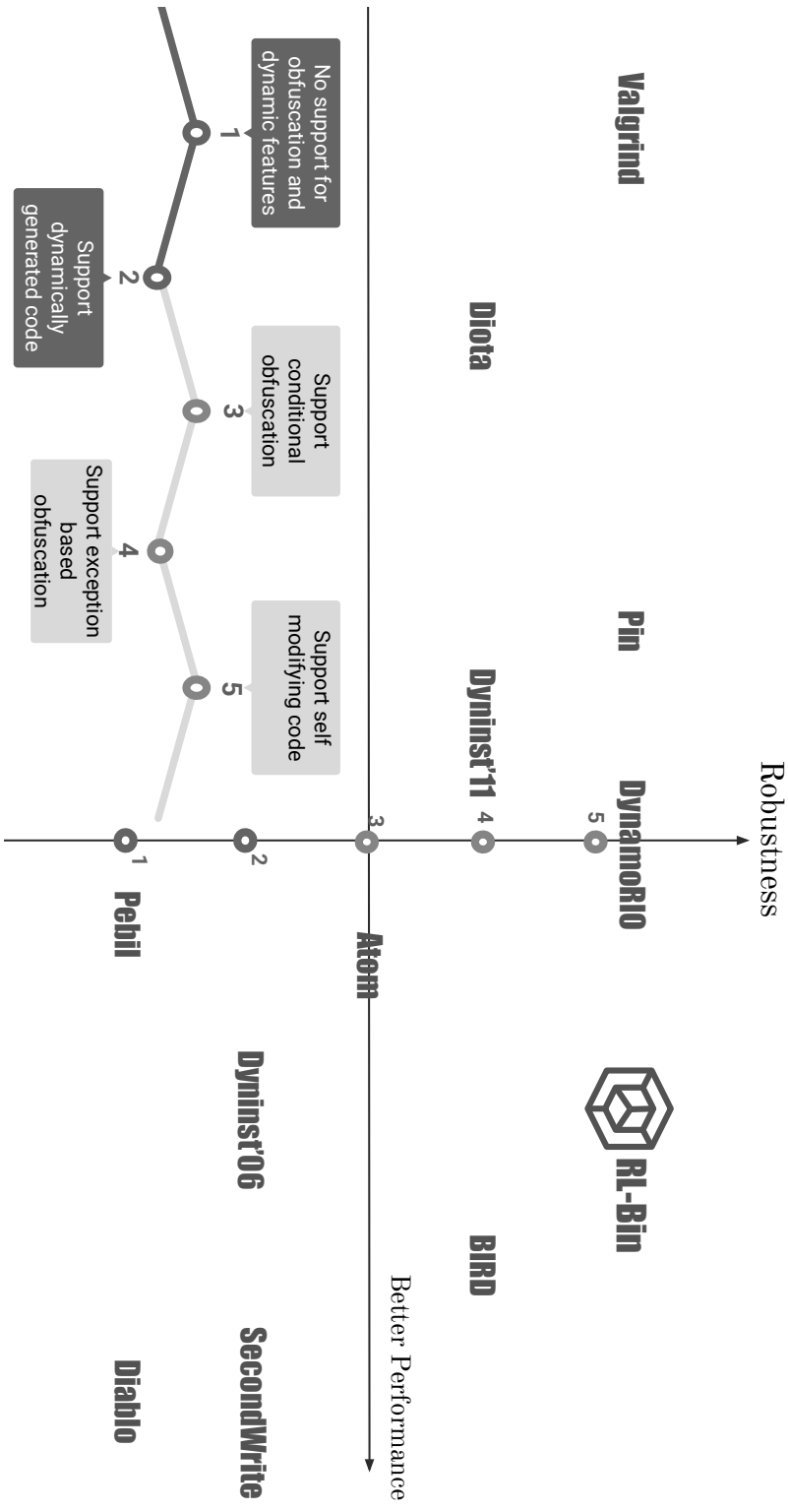


Figure 10.2: Comparison of Robustness and Performance of Static and Dynamic Rewriters

Figure 10.2 illustrates and compares the robustness and run-time overhead of the static and dynamic rewriters that we discussed throughout this chapter. As it can be seen, RL-Bin is both robust and also has low run-time overhead which makes it the only practical choice for instrumenting binaries in live deployment.

10.3 Deobfuscation Tools

Research survey papers such as [93, 94] extensively study and discuss obfuscation tools and mitigation techniques. In this subsection, however, we only briefly go over some of the solutions proposed to counteract obfuscation. Specifially, we focus on low-overhead solutions that are robust and capable of deobfuscating a wide range of binaries.

The first type of deobfuscation tools attempts to unpack the encrypted code bytes. Some of the static unpacking tools use the X-Ray technique [95], which analyzes the packed code's statistical properties to determine the encryption algorithm used. The downside of these tools is that they are not effective against advanced encryption techniques. The other static methods [96, 97] find and reverse the code that does the unpacking automatically. Although this approach has been useful for specific packer tools, it has failed to show effectiveness for a wide range of real-world binaries. [98] uses portable execuatable static features to detect obfuscation of packer tools.

Dynamic unpacking tools are more robust. Fine-grained approaches [99, 100, 101, 102] use whole system emulators to inspect memory write instructions to find

the memory locations that are written to and then executed. The overhead of these methods is overwhelming, and they increase the run-time of the binary by order of magnitudes. There are coarse-grained dynamic approaches [103, 104] that utilize the operating system to track memory writes and execution at page-level granularity. However, these methods cannot identify the exact code bytes that were unpacked and executed.

Another category of deobfuscation tools aims to address anti-disassembly techniques. To handle obfuscation and the issues regarding non-returning calls, [105] uses a modified version of recursive traversal with no assumption about the call instructions that must return. Instead, it will look for specific statistical properties in the byte codes after the call instruction to determine whether they form actual sequences of valid instructions. While this method helps with code coverage, it has not been sufficient for real-world obfuscated code. To handle obfuscated indirect CTIs and increase code coverage, some disassembly tools deploy value-set analysis [106] to find all possible destinations for indirect CTIs and limit the number of targets taken during run-time. However, due to multiple obfuscation techniques applied simultaneously, the analysis of these tools would mostly be incomplete and inconclusive.

The primary method to address anti-rewriting techniques such as self checksumming is redirecting memory read to an unchanged copy of the modified memory locations. Some solutions [107] achieve this by changing the Operating System handling of Translation Lookaside Buffer (TLB) to provide different versions of cached memory for code and data. Another solution [108] emulates memory read

instructions by using virtualization, which would lead to an order of magnitudes for overhead, preventing these methods from being deployed in live systems.

Chapter 11: Conclusion and Future Works

11.1 Achievements

A binary rewriter is a software tool that can change binary code (also known as machine code) without needing its source code to improve it in some way, such as in its security, performance, manageability, or track-ability. Binary programs are widespread today in IP-protected applications meant for distribution to customers and high-performance codes. Conversations with industry professionals have revealed that they will not accept an unreliable tool that may occasionally crash the program. Nor will they accept a tool that has more than a few percent overheads in deployment use. No existing rewriter can fully meet these requirements. Thus all the advantages of instrumentation and monitoring in security, performance, manageability, and track-ability are lost for deployed binary programs.

The goal of this thesis was to gain the benefits of instrumentation and modification for binary code. To this end, we needed to develop a binary rewriter that fulfills two main criteria: First, it must work for a different type of binaries, including those produced by commercial compilers from a wide variety of languages, and possibly modified by obfuscation tools. Second, the binary rewriter must be low overhead.

We have demonstrated in this thesis that we have achieved the goal of robustness. RL-Bin is robust, meaning that it can handle all types of obfuscation and other troublesome features that exist in benign binaries. Chapters 5 and 6 have described these features and our countermeasures in detail.

In addition, RL-Bin has only 4% run-time overhead for the SPECrate benchmark suite, which is low enough for practical deployment in live systems. This is achieved by using multiple optimization methods, including a Just-In-Time (JIT) dynamic analysis of the discovered code and traditional data flow analysis concepts, to find "Safe" functions and reduction of overhead by eliminating redundant checks. In addition, we have in-depth analysis and exploration of trade-offs for optimization methods to get the run-time overhead.

The result is the first In-Place dynamic binary rewriter – which does not use a code cache – that combines the robustness and coverage of a dynamic rewriter with the low overhead of a static rewriter.

11.2 Future Works

As future work, it is possible to extend the secure execution tool, described in section 8.2, to protect all types of vulnerable indirect CTIs to increase coverage against other types of attacks. It is also possible to improve the transparency of RL-Bin by employing methods to conceal the presence of RL-Bin to avoid detection by anti-rewriting techniques. There are also plans to improve RL-Bin's Application Programming Interface and extend its support for multiple platforms and operating

systems.

11.2.1 RL-Bin-Based Implementation of CFI and Other Analysis Tool

We imagine that RL-Bin can be used for the full implementation of CFI. To protect all types of vulnerable indirect CTIs to increase coverage against other types of attacks. To this end, one must find dynamic methods to restrict an indirect call or jump's potential destinations. Then, during run-time, RL-Bin added instrumentations to ensure that indirect calls and jumps follow the restricted control flow.

In addition, we believe that RL-Bin can be the basis of numerous run-time tools. The capability to easily monitor control flow enables the user to implement new tracing tools, logging, and code coverage. One crucial distinction is that RL-Bin has adaptability, which means that code analysis instrumentation does not have to be permanent and removed after the goal is achieved. This would further reduce the overhead.

We also believe that RL-Bin can be an appropriate basis for an OS dependant in-process debugger, which can provide fast and flexible debugging infrastructure from within the binary process space.

11.2.2 Improve Transparency

RL-Bin is both robust and has low overhead, but it still has issues with transparency. Some malicious binaries use anti-debugging techniques to exploit the lack of

transparency in instrumented binaries. They detect the presence of a binary rewriter and terminate the execution of the application. In future work, transparency issues would be resolved for sensitive binaries.

11.2.3 Programming Interface Extention

Our application programming interface could be enhanced to allow customization of internal data structures and provide control over other internal aspects of the system. This would enable the user to have greater flexibility when developing binary analysis tools based on RL-Bin. For example, one can extract internal information from within RL-Bin to develop a security tool with less overhead than using our current programming interface.

11.2.4 Support Additional Platforms

The current version of RL-Bin is implemented for x86 architecture and the Windows operating system. We tested RL-Bin with several commercial applications and compiled binaries with different compilers, including GCC, MSVS, and ICC. RL-Bin has also been tested with heavily obfuscated code.

As many applications are developed for other architectures and operating systems, as future work, RL-Bin will add support for x-64 and ARM architectures and support Linux OS. This would further increase the usability of RL-Bin as a basis for developing binary analysis tools.

11.3 Summary

We have developed a new type of binary rewriter called RL-Bin that can reliably rewrite all benign programs and incurs low run-time overhead. It does so using a design that

1. avoids the copying and address translation inherent in code-cached-based dynamic rewriters by rewriting the memory image in-place;
2. is purely dynamic and continuously instruments the code to conceptually monitor every control transfer to discover new code;
3. rewrites a memory block in the code segment only after it is known to be code at run-time;
4. uses a design that adaptively removes code-discovering instrumentation at run-time after it is no longer needed; and
5. uses just-in-time (JIT) analysis to perform further optimizations to reduce overhead.

The implementation of RL-Bin is robust and low overhead and is well-tested. RL-Bin's design and optimization methods have empowered RL-Bin to rewrite binaries with very low run-time overhead (1.04x on average for SPECrate 2017) and comparatively low memory overhead (1.69x for SPECrate 2017). In comparison, other dynamic rewriters have a high run-time overhead (1.16x for DynamoRIO,

1.29x for Pin, and 1.20x for Dyninst) and have a bigger memory footprint (2.5x for DynamoRIO, 2.73x for Pin, and 2.3x for Dyninst).

Bibliography

[1] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248, 2012.

[2] Yougang Song and Brett D Fleisch. Utilizing binary rewriting for improving end-host security. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1687–1699, 2007.

[3] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566. IEEE, 2017.

[4] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[5] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 299–308. ACM, 2012.

[6] Piotr Bania. Securing the kernel via static binary rewriting and program shepherding. *arXiv preprint arXiv:1105.1846*, 2011.

[7] Matteo Rizzo, Chemin de la Praz, and Julien Voisin. Hardening and testing privileged code through binary rewriting. 2020.

[8] Yikun Hu, Yuanyuan Zhang, and Dawu Gu. Automatically patching vulnerabilities of binary programs via code transfer from correct versions. *IEEE Access*, 7:28170–28184, 2019.

[9] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *ACM SIGARCH Computer Architecture News*, 35(2):482–493, 2007.

[10] David Williams-King and Junfeng Yang. Codemason: Binary-level profile-guided optimization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 47–53, 2019.

[11] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.

[12] Chad M Huneycutt, Joshua B Fryman, and Kenneth M Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *Proceedings International Conference on Parallel Processing*, pages 621–630. IEEE, 2002.

[13] Alexis Engelke, David Hildenbrand, and Martin Schulz. Optimizing performance at runtime using binary rewriting.

[14] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*. Citeseer, 2001.

[15] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54. IEEE, 2002.

[16] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, 2017.

[17] Ulfar Erlingsson and Fred B Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 287–295. IEEE, 2000.

[18] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[19] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper van Woudenberg. Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection. *arXiv preprint arXiv:2011.14067*, 2020.

[20] Yongjun He, Hui Shu, and Xiaobing Xiong. Protocol reverse engineering based on dynamorio. In *2009 International Conference on Information and Multimedia Technology*, pages 310–314. IEEE, 2009.

[21] Juan Caballero and Dawn Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and natrewriting. *CyLab*, page 32, 2007.

[22] Marek Olszewski, Jeremy Cutler, and J Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[23] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[24] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.

[25] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[26] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference*, pages 401–414, 2020.

[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[28] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, volume 26, pages 27–40, 2012.

[29] Pankaj Kohli and Bezawada Bruhadeshwar. Formatshield: A binary rewriting defense against format string attacks. In *Australasian Conference on Information Security and Privacy*, pages 376–390. Springer, 2008.

[30] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. *ACM SIGARCH Computer Architecture News*, 41(1):317–328, 2013.

[31] Amitabha Roy, Steven Hand, and Tim Harris. Hybrid binary rewriting for memory access instrumentation. *ACM SIGPLAN Notices*, 46(7):227–238, 2011.

[32] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally A Mckee, Bronis R De Supinski, and Andy Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):12–es, 2007.

[33] Marwa Yusuf, Ahmed El-Mahdy, and Erven Rohou. Runtime on-stack parallelization of dependence-free for-loops in binary programs. *IEEE Letters of the Computer Society*, 2(1):1–4, 2019.

[34] Rajeev Kumar Barua and Aparna Kotha. Automatic parallelization using binary rewriting, February 4 2014. US Patent 8,645,935.

[35] Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse. Feasibility of dynamic binary parallelization. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, 2011.

[36] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 64–73. IEEE, 2011.

[37] Haibo Zhao, Fei Zheng, Jian Wu, Baosong Nan, Boliang Li, and Kuizhi Mei. Automatic parallelization for binary on multi-core platforms. In *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, pages 1–6, 2018.

[38] Toshi Piazza. *Real-time Address Leak Detection on the Dynamorio Platform Using Dynamic Taint Analysis*. Rensselaer Polytechnic Institute, 2018.

[39] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.

[40] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*, pages 749–754. IEEE, 2006.

[41] Dave Shackleford. A new era in endpoint protection. `https://go.crowdstrike.com/rs/281-OBQ-266/images/ReportSANSProductReview.pdf`, 2017. retrieved: October, 2019.

[42] Derek L Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[43] Binary obfuscation project tool. `https://www.codeproject.com/Articles/856846/Binary-Obfuscation`. retrieved: October, 2019.

[44] Arxan binary obfuscation. `https://www.arxan.com/technology/obfuscation/`. retrieved: October, 2019.

[45] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[46] Anna R Karlin, Mark S Manasse, Larry Rudolph, and Daniel D Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

[47] SA Hex-Rays. Ida pro disassembler, 2008.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[49] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. Toward the deconstruction of dyninst. *Comput. Sci. Dept., Univ. Wisconsin, Madison, Tech. Rep.*, 2007.

[50] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673. IEEE, 2015.

[51] Markus FXJ Oberhumer. Upx the ultimate packer for executables. *http://upx. sourceforge. net/*, 2004.

[52] Pecompact, an advanced windows executable compressor for use by software developers and vendors. `https://bitsum.com/pecompact.htm`. retrieved: 2020.

[53] Asprotect, a multifunctional exe packing tool designed to protect 32-bit applications. `http://www.aspack.com/asprotect32.html`. retrieved: 2020.

[54] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[55] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[56] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[57] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[58] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[59] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[60] Steve Christey, J Kenderdine, J Mazella, and B Miles. Common weakness enumeration. *Mitre Corporation*, 2013.

[61] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 12–pp. IEEE, 2006.

[62] Craig Macdonald, Iadh Ounis, and Ian Soboroff. Overview of the trec 2007 blog track. In *TREC*, volume 7, pages 31–43, 2007.

[63] Ollydbg ver 1.10. `http://www.ollydbg.de/`. retrieved: October, 2019.

[64] Microsoft corporation debugging tool for windows. `https://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx`. retrieved: October, 2019.

[65] Vasileios Hatzivassiloglou and Kathleen R McKeown. Predicting the semantic orientation of adjectives. In *Proceedings of the 35th annual meeting of the association for computational linguistics and eighth conference of the european chapter of the association for computational linguistics*, pages 174–181. Association for Computational Linguistics, 1997.

[66] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198. IEEE, 2019.

[67] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[68] Martin Dimitrov and Huiyang Zhou. Anomaly-based bug prediction, isolation, and validation: An automated approach for software debugging. *SIGARCH Comput. Archit. News*, 37(1):61–72, March 2009.

[69] Bryan Buck and Jeffrey K Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[70] Dynamorio api. `http://dynamorio.org/docs/API_BT.html`. retrieved: October, 2019.

[71] Intel pin api. https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/group__API__REF.html. retrieved: October, 2019.

[72] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):49, 2019.

[73] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 295–308. ACM, 2013.

[74] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pages 25–25. USENIX Association, 1995.

[75] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[76] Shuai Wang, Pei Wang, and Dinghao Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 236–247. IEEE, 2016.

[77] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 129–140, 2014.

[78] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *PLDI*, pages 151–163, 2020.

[79] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 201–, Washington, DC, USA, 1997. IEEE Computer Society.

[80] Squeeze project weboage. https://www2.cs.arizona.edu/projects/squeeze/. retrieved: 2020.

[81] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 2005.

[82] Amitabh Srivastava. A practical system for intermodule code optimization at link-time. *Journal of programming Languages*, 1(1):1–18, 1993.

[83] Robert Muth, Saumya K Debray, Scott Watterson, and Koen De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software: Practice and Experience*, 31(1):67–101, 2001.

[84] Robert Cohn, David Goodwin, P Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for alpha/nt executables. In *USENIX Windows NT Workshop*, pages 17–24, 1997.

[85] Robert S Cohn, David W Goodwin, P Geoffrey Lowney, and N Rubin. Optimizing alpha executables on windows nt with spike. *Digital Technical Journal*, 9:3–20, 1997.

[86] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 7–12. IEEE, 2005.

[87] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):5–es, 2007.

[88] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled El-Wazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic obfuscation in malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 97–118. Springer, 2017.

[89] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[90] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, pages 22–es, 2004.

[91] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16. ACM, 2011.

[92] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[93] Kevin A Roundy and Barton P Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, 46(1):1–32, 2013.

[94] Hassen Saıdi, Phillip Porras, and Vinod Yegneswaran. Experiences in malware binary deobfuscation. *Virus Bulletin*, 2010.

[95] Frédéric Perriot and Peter Ferrie. Principles and practise of x-raying. In *Virus Bulletin Conference*, pages 51–56, 2004.

[96] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In *2009 16th Working Conference on Reverse Engineering*, pages 167–176. IEEE, 2009.

[97] Saumya Debray and Jay Patel. Reverse engineering self-modifying code: Unpacker extraction. In *2010 17th Working Conference on Reverse Engineering*, pages 131–140. IEEE, 2010.

[98] Dhruwajita Devi and Sukumar Nandi. Pe file features in detection of packed executables. *International Journal of Computer Theory and Engineering*, 4(3):476, 2012.

[99] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, 2007.

[100] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, 2008.

[101] Zhanyong Tang, Kaiyuan Kuang, Lei Wang, Chao Xue, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, Jie Liu, and Zheng Wang. Seead: A semantic-based approach for automatic binary code de-obfuscation. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 261–268. IEEE, 2017.

[102] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Daniel Xiapu Luo. Automated deobfuscation of android native binary code. *arXiv preprint arXiv:1907.06828*, 2019.

[103] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[104] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 431–441. IEEE, 2007.

[105] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[106] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[107] Glenn Wurster, Paul C Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005.

[108] Nathan E Rosenblum, Gregory Cooksey, and Barton P Miller. Virtual machine-provided context sensitive page mappings. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, 2008.