The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

ADVANCED REVERSE ENGINEERING TECHNIQUES FOR BINARY CODE SECURITY RETROFITTING AND ANALYSIS

A Dissertation in
Information Sciences and Technology
by
Shuai Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2018

The dissertation of Shuai Wang was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Sencun Zhu
Associate Professor of Computer Science and Engineering & Information Sciences and Technology

Trent Jaeger
Professor of Computer Science and Engineering

Danfeng Zhang
Assistant Professor of Computer Science and Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

In software security, many techniques and applications depend on binary code reverse engineering, i.e., analyzing and retrofitting executables with the source code unavailable. Despite the fact that many security hardening techniques rely heavily on reverse engineering, modern binary disassembling and reconstruction techniques still cannot adequately fulfill many of the requirements. In particular, no reverse engineering tool can disassemble an executable into assembly code which can be *reassembled* back in a fully automated manner, especially when the processed objects are Commercial-Off-The-Shelf (COTS) binaries with most symbol and relocation information stripped. Due to the lack of support for direct reassembling, existing binary instrumentation tools leverage patch or replica-based rewriting techniques to guarantee the correct functionality of the instrumented outputs, which usually incur high execution slowdown and binary code size increase.

We present UROBOROS, a tool that can disassemble legacy executables to the extent that the generated code can be assembled back to working binaries without manual effort. The key technique proposed in UROBOROS is named *reassembleable disassembling*, in which we develop a set of methods to precisely recover each component of a binary executable, including code, data and meta-information. In particular, UROBOROS is the first to be capable of not only recovering the assembly program, but enabling *reassembling* of the disassembled output with the correct functionality. We further extend UROBOROS into a general purpose binary instrumentation platform with a rich set of binary instrumentation APIs and utilities. Our evaluation on widely-used program binaries shows that UROBOROS can provide support for reassembly and instrumentation on legacy binary executables with better performance, lower labor cost, and a broader scope of applications.

In addition, we build advanced binary analysis and instrumentation applications for security purpose. Function recognition in program binaries serves as the foundation for many security retrofitting and analysis tasks. However, as binaries are usually stripped before distribution, function information is indeed absent in most binaries. We develop FID to recognize functions through machine learning

techniques. FID extracts semantic information from binary code and trains a machine learning model for recognition. Our evaluation demonstrates that FID has a high recognition accuracy on commonly-used program binaries as well as obfuscated code.

We further build program diversification tools. By transforming software into different forms before deployment, software diversification can effectively mitigate many attacks. Enlightened by research in other areas, we seek to apply different diversifications to the same program for a synergy effect such that the resulting hybrid transformations can have boosted diversification effects at modest cost. Given a set of commonly-used diversification passes, we propose a novel selection strategy to promptly construct a transformation composition that performs better than any single transformation in the set.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost I would like to express my special gratitude and appreciation to my Ph.D. advisor Dinghao Wu. I want to thank him for encouraging my research and providing invaluable advice all the time. He has been so supportive since the first day I started to work in his lab. He gives me tremendous helpful advices on research and career development along the whole journey. During my Ph.D. pursuit, he always gave me the freedom I needed to explore whatever I was interested, at the same time continuing to contribute his support, feedback and encouragement. I cannot image to have a better adviser for my Ph.D. study.

I would also like to thank Peng Liu, Sencun Zhu, Trent Jaeger, and Danfeng Zhang for serving as my thesis committee members. I strongly appreciate their time and valuable feedbacks on my thesis drafts. I would particularly acknowledge Danfeng Zhang, who has been collaborating with me on research projects that lead to very fruitful results for the recent two years.

I am most grateful to my parents for their love, sacrifices, and support. Without them, this thesis would never have been written. I also want to give many thanks to all my friends at Penn State University, especially my labmates. For the past five years we have established great relationships and shared lots of wonderful times together.

This research was supported in part by the National Science Foundation (NSF) under grants CNS-1223710, CCF-1320605, and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

Chapter 1 Introduction

Reverse engineering has many important applications in computer security, one of which is retrofitting software for safety and security hardening when source code is not available. However, we surprisingly found that no existing tool is able to disassemble executable binaries into assembly code that can be *correctly assembled back* in a fully automated manner, even for simple programs. Actually in many cases, the resulted disassembled code is far from a state that an assembler accepts, which is hard to fix even by manual effort. This has become a severe obstacle. People have tried to overcome it by patching or duplicating new code sections for retrofitting of executables, which is not only inefficient but also cumbersome and restrictive on what retrofitting techniques can be applied to.

In this chapter, we first survey existing binary reverse engineering tools and reveal our surprising finding that all the disassemblers share one common limitation (i.e., the lack of reassembility). We then seek to design binary disassemblers from a new angle; executables are disassembled to the extent that the generated code can be assembled back to working binaries without manual effort. We name the new disassembling technique as reassembleable disassembling.

1.1 Limitations of Existing Binary Reverse Engineering Tools

In computer security, many techniques and applications depend on binary reverse engineering, i.e., analyzing and retrofitting software binaries with the source code unavailable. For example, software fault isolation (SFI) [1–5] rewrites untrusted

programs at the instruction level to enforce certain security policies. To ensure program control-flow integrity (CFI, meaning that program execution is dictated to a predetermined control-flow graph) [6–11] without source code, the original control-flow graph must be recovered from a binary executable and the binary must be retrofitted with the CFI enforcement facility embedded [12, 13]. Symbolic taint analysis [14] on binaries must recover assembly code and data faithfully. The defense techniques against return-oriented programming (ROP) attacks also rely on binary analysis and reconstruction to identify and eliminate ROP gadgets [15–19].

Despite the fact that many security hardening techniques are highly dependent on reverse engineering, flexible and easy-to-use binary manipulation itself remains an unsolved problem. Current binary decompilation, analysis, and reconstruction techniques still cannot fully fulfill many of the requirements from downstream. To the best of our knowledge, there is no reverse engineering tool that can disassemble an executable into assembly code which can be *reassembled* back in a fully automated manner, especially when the processed objects are commercial-off-the-shelf (COTS) binaries with most symbol and relocation information stripped.

We have investigated many existing tools from both the industry and academia, including IDA Pro [20], Phoenix [21], Dagger [22], MC-Semantics [23], Second-Write [24], BitBlaze [25], and BAP [26]. Unfortunately, these tools focus more on recovering as much information (e.g., data and control structures) as possible for analysis purposes, but less on producing assembly code that can be readily assembled back without manual effort. Hence, none of them provide the desired disassembly and reassembly functionality that we consider, even if the processed binary is small and simple.

Due to lack of support from reverse engineering tools, people build high-level security hardening applications based on partial binary retrofitting techniques, including binary rewriting tools such as Alto [27], Vulcan [28], Diablo [29], and binary reuse tools such as BCR [30] and TOP [31]. We consider binary rewriting as a partial retrofitting technique because it can only instrument or patch binaries, thus not suitable for program-wide transformations and reconstructions. As for binary reuse tools, they work by dynamically recording execution traces and combining the traces back to an executable, meaning the new binary is only an incomplete part of the original binary due to the incomplete coverage of dynamic program analysis.

Partial retrofitting has notable drawbacks and limitations:

- Patch-based rewriting could introduce non-negligible runtime overhead. Since the patch usually lives in an area different from the original code of the binary, interactions between the patch and the original code usually require a large amount of control-flow transfers.
- Patch-based rewriting usually relocates instructions at the patch point to somewhere else to make space for the inserted code. As a result, it requires the affected instructions to be relocatable by default.
- Instrumentation-based rewriting expands binary sizes significantly, sometimes generating nearly double-sized products.
- Binary reuse often requires a binary component to be small enough for dynamic analysis to cover; otherwise the correctness cannot be guaranteed.

1.2 Reassembleable Disassembling

Having investigated previous research on binary manipulation and reconstruction, we believe that it could be a remarkable improvement if we are able to automatically recover the assembly from binaries and make the assembly code ready for reassembly. When a binary can be reconstructed from assembly code, many high-level and program-wide transformations become feasible, leading to new opportunities for research based on binary retrofitting such as CFI, diversification, and ROP defense.

Our goal is quite different from previous reverse engineering research. Instead of trying to recover high-level data and control structures from program binaries which helps binary code analysis, we aim at a more basic objective, i.e., producing assembly code that can be readily reassembled back without manual effort, which we call the *reassembility* of disassembly. Although the research community has made notable progress on binary reverse engineering, reassembility is still somewhat blank due to the lack of attention. In this sense, our contribution is complementary to existing work.

With that said, we believe that the technical challenge is also a cause of the deficiency in binary reassembly support from existing tools. We have confirmed that the key to reassembility is making the assembly code *relocatable*. Relocation is a

linker concept, which is basically for ensuring program elements defined in different source files can correctly refer to each other after being linked together. Being relocatable is also a premise for supporting program-wide assembly transformations. In COTS binaries, however, the information necessary for making disassembly results relocatable is mostly unavailable. There has been research trying to address the relocation issue [30–32], but existing work mostly relies on dynamic analysis which is unlikely to cover the whole program.

We present UROBOROS, a disassembler that does reassembleable disassembling. In UROBOROS, we develop a set of methods to precisely recover each part of a binary executable. In particular, we are the first to be capable of not only recovering code, but also data and meta-information from COTS binaries without manual effort. We have implemented a prototype of UROBOROS and tested it on 244 binaries, including the whole set of GNU Coreutils and the C programs in SPEC2006 (including both 32-bit and 64-bit versions). In our experiments, most programs reassembled from UROBOROS's output can pass functionality tests with negligible execution overhead, even after repeated disassembly and reassembly. Our preliminary study shows that UROBOROS can provide support for program-wide transformations on COTS binaries.

In summary, we make the following contributions:

- We initiate a new focus on reverse engineering. Complementary to historical work which mostly focuses on recovering high-level semantic information from binary executables or providing support for binary analysis, our work seeks to deliver *reassembility*, meaning we disassembles binaries in a way that the disassembly results could be directly assembled back into working executables, without manual edits.

- We identify the key challenge is to make the disassembled program *relocatable*, and propose our key technique to recover references among immediate values in the disassembled code, namely "symbolization".

- With reassembility, our research enables direct binary-based transformation without resort to the previously used *patching* method, and can potentially become the foundation of binary-based software retrofitting.

- We implement a prototype of UROBOROS and evaluate its strength on binary reassembly. We applied our technique to 244 binaries, including the whole set of GNU Coreutils and SPEC2006 C binaries. The experiment results show that our tool does correct disassembly and introduces only modest cost.

- Our disassembler produces "normal" assembly code in the sense that binaries reassembled from UROBOROS's output assembly can again be disassembled (and hence the name UROBOROS[1]), or be used to accomplish other reverse engineering tasks. We verify this by repeating the disassemble-reassemble loop for thousands of times on different binaries.

- We further extend UROBOROS into a general purpose binary instrumentation platform with a rich set of binary instrumentation APIs and utilities.[2] We perform comparative evaluations between UROBOROS and the state-of-the-art binary instrumentation tools. To demonstrate the versatility of UROBOROS, we also implement two real-world reverse engineering tasks which could be challenging for other instrumentation tools to accomplish. Our experimental results show that UROBOROS outperforms the existing binary instrumentation tools with better performance, lower labor cost, and a broader scope of applications.

1.3 Thesis Organization

The rest of the thesis is organized as follows. We first present an overview of the related research work in Chapter 2. Chapter 3 proposes a novel disassembling technique of binary executables, reassembleable disassembling. Chapter 4 elaborates on the design and implementation of a generic binary instrumentation platform UROBOROS that implements our reassembleable disassembling algorithm. We further proposes two binary instrumentation applications on top of UROBOROS in Chapter 5 and Chapter 6 for function recognition and software diversification. We present further discussion in Chapter 7 and conclude the thesis in Chapter 8.

[1] Uroboros is a symbol depicting a serpent eating its own tail.

[2] UROBOROS is publicly available for download from `https://github.com/s3team/uroboros`.

Chapter 2
Related Work

In this chapter, we first review related research on binary code reverse engineering and instrumentation. We then review research articles on binary analysis and retrofitting applications such as software diversification and function recognition. Literatures reviewed in this section are actually related and also inspired this thesis research.

2.1 Reverse Engineering and Binary Instrumentation

2.1.1 Binary Disassembling

There is no disassembler known to us that can generate working assembly code from binaries whose symbol and relocation information is stripped. IDA Pro [20] is considered as the best commercial disassembler available on the market. It can decode binaries into assembly and further decompile assembly into C code for program analysis. However, the assembly code produced by IDA Pro cannot be directly used as the input of any assembler. As stated in its manual [33], assembly code produced by IDA Pro is meant for analysis and cannot be directly reassembled or recompiled.

SecondWrite [24] leverages multiple static analysis techniques to lift binaries into LLVM IR. It is reported that the recovered LLVM IR can be converted back into C code given the LLVM's IR-to-C backend. However, it is unclear to us how SecondWrite symbolizes the data sections and recovers the meta-data information of the binaries. The paper does not contain an evaluation on this recompilation

functionality. Moreover, the IR-to-C backend has been removed from LLVM release since 3.1, because it is not mature enough to handle non-trivial programs [34].

Dagger [22] is another tool that translates native code into LLVM IR, but the implementation is far from complete. There is a pre-release version available online. We tried to use it to decompile a simple binary (compiled from a C program with only empty main function). The decompiler reported several errors and generated an LLVM IR file which cannot be compiled back into binary due to lack of some symbol definitions.

MC-Semantics [23] is yet another tool for native code to LLVM IR translation. We used MC-Semantics to decompile some quickly written mini programs. Although the code produced by MC-Semantics can be made binaries, the execution results of these binaries are not the same as the originals, which we believe is due to incorrect symbol references. In addition, different from previously reviewed work, MC-Semantics works at the scale of object files rather than executables. Lacking the ability to handle linked binary programs narrows its scope of application.

BAP [26] is a binary analysis platform that comes with a disassembler. It can lift assembly code to a BAP-defined high-level intermediate representation that can be further analyzed statically. Several reverse engineering tools have been built based on BAP, including the C type recovery tool TIE [35] and the C control-flow recovery tool Phoenix [21]. Although BAP provides solid support for binary analysis, the strength of its disassembler is also limited to analysis only.

There could be multiple reasons that existing tools fail on reassembling. One reason is the technical challenges such as separating code and data, symbolizing the data sections, etc. The other reason could be the difference in the design goals. Most existing tools aim to produce more readable code or code that can be analyzed, not for the purpose of translation and reassembly. We emphasize that the ability to reassemble the output from a disassembler can provide an enabling infrastructure, facilitating further research.

2.1.2 Static Binary Instrumentation

Static methods instrument the whole input binary before execution [24, 29, 36–38]. It has been widely used in security hardening tasks such as control-flow hijacking mitigation [17], software control-flow integrity enforcement [12, 13], and retrofitting

security defenses [39]. However, due to the difficulties of disassembly, most previous static binary rewriting tools have to require relocation or debug information [29, 36, 38]. SecondWrite [24] performs advanced static analysis to lift binary code into LLVM IR. The IR is then employed for binary rewriting. PSI [37] supports robust security-related binary instrumentation through binary rewriting.

As we have pointed out, a common feature of conventional static binary instrumentation is that it relies on binary rewriting. It has to carefully relocate instructions at the instrumentation point to arrange space for newly inserted code. To this end, patch-based and replica-based instrumentations are frequently used. However, the newly generated binary could exhibit high execution slowdown, size increase, and even error functionality. The key feature of UROBOROS is that it does not rely on binary rewriting. Instead, it leverages the advanced disassembling technique [40] to directly inline the instrumentation code into the target binary. Therefore, UROBOROS delivers a decent runtime performance and a small increase in code size.

2.1.3 Dynamic Binary Instrumentation

Dynamic binary instrumentation inserts additional code when a program executes, which is more accurate than static binary instrumentation since it only considers the real path taken at run time [41–43]. Dynamic binary instrumentation has been widely used for program performance profiling [44] and security-oriented execution monitoring tasks [14, 45]. Pin [41] and DynamoRIO [42] undertake lightweight instrumentation jobs, while Valgrind [43] is designed for more heavy-weight instrumentation tasks, e.g., memory debugging. Among them, Pin is widely used for goal-driven binary security tasks, such as dynamic taint analysis [14, 45]. DynInst [46, 47] supports both static and dynamic binary instrumentation. It disassembles the stripped binaries and instruments them statically or dynamically. Dynamic instrumentation methods cannot be deployed in some scenarios such as real-time or mission-critical systems due to the runtime instrumentation environment.

2.2 Applications

2.2.1 Binary Code Reuse

Binary reuse is mostly based on dynamic analysis. One of the representative binary reuse tools is BCR [30]. BCR extracts and reuses functions from binaries with a hybrid approach. BCR first executes binaries in a monitored environment and records execution traces and memory dumps. Binaries are then statically disassembled starting from the entry point. In the disassembly process, the dynamically collected information is used to resolve the destinations of indirect branches. In the end BCR manages to extract a "closure" of code reachable from the entry point which can be reused by other programs. Clearly, the correctness of the reused code cannot be guaranteed if BCR does not cover all feasible execution paths.

In addition to BCR, there are other binary reuse tools that employs similar basic ideas, such as Inspector Gadget [32] and TOP [31]. While these tools have made improvements in different aspects, the fact that they all rely on dynamic analysis leads to the incompleteness issue, more or less. In general, these tools can only do partial binary retrofitting.

2.2.2 Software Diversification

The idea of software diversification has been studied for decades. Diversifying approaches are presented with various scopes from a single instruction to the whole program. Fine-grained approaches such as instruction and basic block-level diversification aim at diversifying instructions within one basic block or sequences of basic blocks. Typical transformations include dead code insertion, instruction substitution, and basic block reordering [48–50]. These transformations have been adopted by both malware triage evasion [51, 52] and program randomization [17, 19]. Coarse-grained approaches are essentially deployed in the program runtime environment, hardening the program context from being exploited. Stack-layout randomization [53] and address space layout randomization (ASLR) [54] can deploy probabilistic defense, say, the unpredictable memory addresses can effectively impede code reuse attacks. However, attacks are still feasible due to limited randomization space of these coarse-grained approaches [55].

Other related work propose to randomize the encoding of program instructions [56]. Program encoding leverages one reversible encoding method to statically translate program text into encoded data. Usually a decoding routine and the decoding key is distributed inside the program, which can decode the data later. The encoded program can defeat a large number of static analyses. Improved techniques like virtual machine packer have been utilized to construct more secure binaries [57–60].

Even though a few research work have touched the process of "composite diversification" [48, 61, 62], they are limited to some primary ideas or the proposed approaches only transform input program with very limited iterations. To our best knowledge, there are not existing work undertake an in-depth study on the composite transformation synergy.

2.2.3 Function Recognition in Binary Executables

Function identification is considered as the foundation for many binary analysis and test applications, and there has been a number of related work in this topic [63–67]. Rosenblum et al. [65] propose to use machine learning based approach to address the function recognition problem. Bao et al. [68] detail multiple challenges in this topic, and propose a weighted prefix-tree based approach to train the recognition model. As aforementioned, their work have two implementations by learning both machine code bytes and instruction sequences, and it has been evaluated that ByteWeight can significantly outperform previous work [65] regarding recognition accuracy and processing time. Shin et al. [69] proposes a deep-learning based technique to recognize functions by learning from machine code bytes. It is reported that their work has better performance and less processing time than ByteWeight. Williams et al. [70] report that by extending machine learning based approach with control flow reconstruction, their work can also have comparable performance.

While some recent research work reports good performance using data mining approaches, they essentially share a similar design choice, i.e., they mainly capture the *syntax-level* information to learn. Conceptually, while syntax can be easily extracted, the learned model suffers from syntax changes, e.g., binary *obfuscation* and *diversification*.

Chapter 3
Foundation: Reassembleable Disassembling

Binary executables widely exist in the wild for benign and malicious purposes, and since high-level program representation is largely absent in executables, binary code analysis has become a central focus and challenge in cybersecurity research. We have briefly discussed the motivation and technical challenges for developing a disassembler which can deliver reassembility in Section 1.2. This chapter presents a *ground-breaking* technique for binary code reverse engineering, which transforms binary code into an analysis and retrofitting-friendly format and achieves reassembility. In this chapter, we first detail the technical challenges in Section 3.1. Section 3.2 elaborates on the key contribution of the proposed technique, and we then present the design and optimization of our technique in Section 3.3 and Section 3.4, respectively. The experimental results are reported in Section 3.5.[1]

3.1 Challenges

In this research, we assume that the binaries to disassemble are stripped COTS binaries, namely binaries without any relocation information or symbols, except those necessary for dynamic linking. We also assume that the binaries are compiled from unobfuscated C programs, without self-modifying features. The target hardware architectures of the binaries are x86 and x64. The binary executable format is the Executable and Linkable Format (ELF).

[1]The work of this chapter is published in the 24th USENIX Security Symposium [40].

3.1.1 Raw Disassembly

In this work, raw disassembly is referred to as the process of parsing the binary form of a program to its raw textual representation. The difficulty of raw disassembly can vary a lot in different situations. In the most general case, this problem is undecidable. One of the reasons is that the problem of statically determining the addresses of indirect jumps is undecidable [71]. Furthermore, the existence of advanced program features such as self-modifying code makes the problem harder. Another issue is that current computer architectures do not distinguish code and data, and there is no easy way for a raw disassembler to distinguish them either. This problem is further worsened by the variable-length instruction encoding used by, for example, the x86 instruction set architecture.

However, with years of intensive effort on improving related techniques, the state of the art can already reach a very high success rate when disassembling binaries compiled from practically legitimate C source code by mainstream compilers. A recent paper by Zhang et al. [12] proposed a novel raw disassembly method which combines two existing disassembly algorithms together. We reimplemented this algorithm and applied it to our evaluation set which includes 244 binaries. No errors were reported by the raw disassembler and subsequent evaluation also verified the correctness of this algorithm on our evaluation set. As a result, we do not consider raw disassembly, or binary decoding, as a major challenge to address in this research.

3.1.2 Reassembly

Successfully decoding the binaries is only the first step to the goal of this research. Ideally, binary reverse engineering tools should be able to support at least the following process:

- The reverse engineering tool disassembles the original binary into assembly code.
- Users can perform static analysis on the disassembled program.
- Users can perform transformations on the disassembled program.

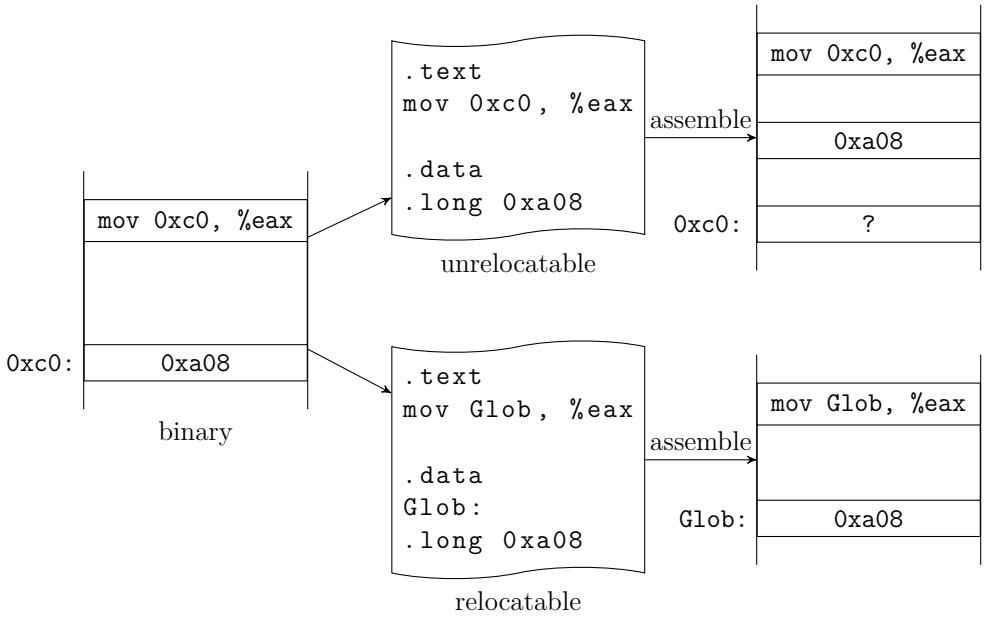


Figure 3.1: Relocatable and unrelocatable assembly code.

- The transformed program can be assembled back into an usable binary executable, with all transformation effects retained.

Although it may not be obvious, the feasibility of the first three steps does not naturally imply the feasibility of the last step. There have been reverse engineering tools or platforms that can (partially) enable the first three steps [25, 26], but support for reassembly is still blank.

As mentioned in the introduction, making the assembly code relocatable is the crux of reassembility. Figure 3.1 is an artificial example comparing relocatable and unrelocatable assembly code. In COTS binaries, information required for making disassembly results relocatable is unavailable. Most program transformations inevitably change binary layouts, but a reverse engineering tool has only very limited control over how the linkers assign memory addresses of the program elements, leading to situations illustrated by Figure 3.1. Note the memory cell located at address `0xc0` in the original memory, which is possibly a global variable. The raw disassembly process does not recognize the concrete value `0xc0` in the code as a reference. Thus when this unrelocatable assembly code is reassembled, the resulting binary will very likely be defective because the content of the memory cell at `0xc0` in the original binary may not be placed the same address in the new

binary. In the relocatable assembly, however, the data originally living at `0xc0` is given a symbolic name, and the concrete address `0xc0` is replaced by a reference to this name. This is why relocatable assembly can be reassembled into a working executable.

As suggested by the example, if a reverse engineering tool seeks to reassemble the transformed assembly code into a working executable, it has to identify program elements whose addresses could possibly change in the new binary, and lift concrete memory addresses referring to them to abstract symbolic references. Obtaining relocatable assembly from a COTS binary is non-trivial because very little auxiliary information in the binary can be utilized to help identify references among concrete values. Essentially, the problem can be generalized as the following: given an immediate value in the assembly code (either in a code section or data section), is it an memory address or a constant? Although this looks like a typical type analysis problem, in the context of binary reassembly it becomes much more challenging. From a static point of view, since most machine assembly languages are untyped, type inference is difficult in the first place. Compared to high-level programming languages, assembly languages lack explicit syntax for denoting procedure boundaries and basic control-flow logic, making static analysis even more difficult. What is worse, many references live in the data sections, some of which are indirectly referred to by the code via numerous reference hops. At present, most proposed program analysis techniques, either static or dynamic, are code oriented, lacking the capability of analyzing the property of a given data chunk. Finally, reassembly has almost zero tolerance for type inference errors, because a single false positive or false negative can place the reassembled binary in a non-functional state.

Solving the relocatable problem in binary disassembly is the main purpose and contribution of this work. In the rest of the paper, we call the process of identifying references among immediate values in the raw assembly the process of "symbolization". To distinguish the concept from the traditional meaning of disassembling, we call our work *reassembleable disassembling* that generates relocatable assembly code.

In addition to relocation information, a full-fledged disassembler also needs to recover some meta information to make the reassembly feasible. Meta-data sections in a binary executable provide information to direct some link-time and runtime

behavior of the program. They should also be recovered properly in order to ensure the reassembled binaries are semantic-equivalent to the originals.

3.2 Symbolization

This section describes the symbolization problem in detail and presents our solution.

3.2.1 Classification

There are four types of symbol references that we need to identify for reassembility. The classification is based on two criteria—where a reference lives and where a reference points. Basically, we divide the binary into two parts, i.e., the code sections and the data sections, whose contents are as suggested by their names. For ELF binaries on Unix-like platforms, typical code sections include `.text` and `.init` etc. Typical data sections include `.data`, `.rodata`, `.bss`, etc. A symbol reference can live in either code sections or data sections, and can point to either code sections or data sections as well, leading to a total of four types. Figure 3.2 is an example showing all four types of symbol references. We give each of them a short name, i.e., c2c, c2d, d2c, and d2d references.

3.2.2 Method

When it comes to solving the symbolization problem, we have considered various potential solutions. Due to the reasons listed in Section 3.1.2, we conclude that no existing program analysis technique can handle the symbolization problem in our special context. Hence, we decide to turn to another direction. In this work, we identify the immediate values which are actually symbol references by applying several matching rules inferred from our study on a large amount of binaries. Although some of these strategies may not seem exciting at the first sight, they work surprisingly well in our evaluation on 244 binaries compiled from C code.

Since we are solving the symbolization problem in an empirical way, the matching strategies are all based on certain assumptions. Depending on whether an assumption is accepted or not, different rules are applied for symbolization. We now introduce the assumptions and the corresponding symbolization strategies.

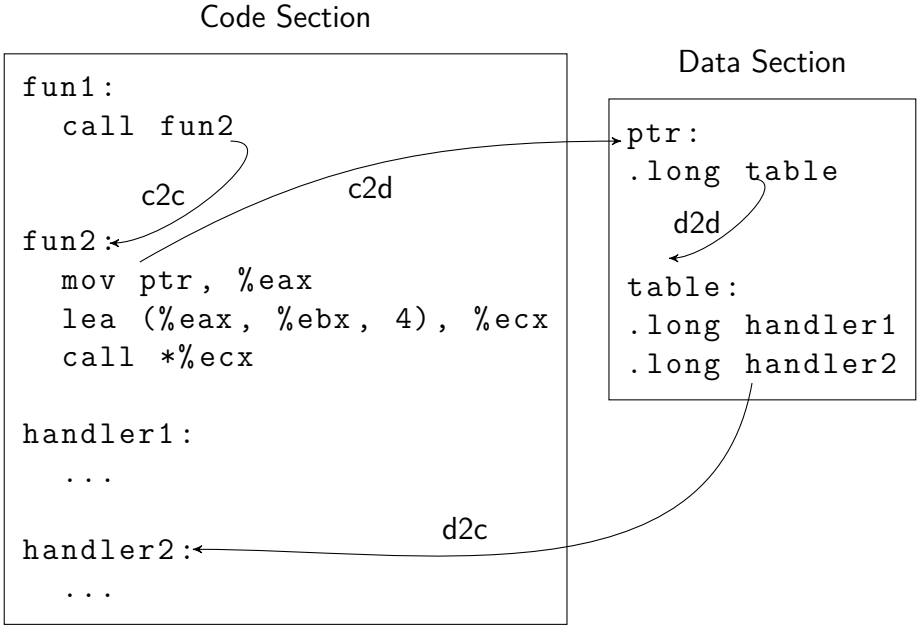


Figure 3.2: Different types of symbol references in assembly code.

At the point of symbolization, we assume that we have already obtained the raw assembly decoded from binaries using the algorithm by Zhang et al. [12], so we can get all immediate values that appear in a binary. There are two kinds of immediate values—constants used as instruction operands and the byte stream living in data sections. Among all these immediate values, some can be excluded from being considered for symbolization at the first place. Unless a program intentionally causes memory access errors, which is rarely the case, an immediate value can be a reference to symbols *only if* this value falls in the address space allocated for the binary. For a binary of reasonable size, the utilization of address space is usually sparse, so there is a wide range of address space which is actually invalid.

Assuming all immediate values are potential symbol references, we can filter out obviously invalid references based on their target addresses. According to our symbol reference classification in Section 3.2.1, a reference can only point to code sections or data sections; especially, if a reference points to code sections, the destination must be the starting address of some instruction. Our study on 244 binaries shows that this simple filter is sufficient to identify c2c and c2d symbol references with full correctness.

The really challenging part is data section symbolization, i.e., identifying d2c and d2d references. The first step of data section symbolization is to slice the data sections, which are continuous areas of binary bytes, into individual values of different lengths. Since the raw disassembly process does not assign the data sections any semantics, there is no ready-made guidance on how they should be sliced. Regarding this problem, we introduce the first assumption which is about binary layout:

> (A1) *All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.*

Since unaligned memory accesses cause considerable performance penalty, compilers tend to keep data aligned by its size. For data alignment, compilers can even sacrifice memory efficiency by inserting padding into data sections. With that said, A1 stays as an assumption because occasionally programmers do want non-aligned data layout. For example, the "packed" attribute supported by GCC allows programmers to override the default alignment settings.

If we accept assumption A1, only n-byte long values which are also n-byte aligned in data sections are considered for symbolization. Alternatively with A1 rejected, all n-byte long memory content in data sections are considered for symbolization. This is implemented as an n-byte sliding window which starts from the beginning of a data section and scans through the entire section in a *first-fit* manner. Each time the sliding window moves forward 1 byte and check the value of the covered bytes. If the value fulfills the basic requirements for being a d2d or d2c reference, it will be considered for symbolization and the sliding window advances n bytes forward. In case that the value does not meet the requirements, the sliding window moves forward 1 byte only.

In addition to assuming the characteristics of binaries, making assumptions on user requirements for our tool also helps improve its performance. As stated earlier, the goal of symbolization is to make assembly code relocatable so that users can perform program-wide transformations on the assembly and then assemble it back to a working executable. From our experience, most transformations on assembly only touch the instructions without modifying the original data. If we make the following assumption

(A2) *Users do not need to perform transformation on the original binary data.*

then we can keep the starting addresses of data sections the same as their old addresses when performing reassembly, by providing a directive script to the linker. In this way, we can ignore d2d references during symbolization simply because we do not need them to be relocatable anymore. Thus, with A2 accepted, only the immediate values that fall within code sections (d2c references) are considered for symbolization. Contrarily without accepting this assumption, immediate values that fall within either code sections (d2c references) or data sections (d2d references) will need to be considered for symbolization.

We want to avoid symbolizing d2d references because they are used in a very flexible manner. On the other hand, there are more common patterns in d2c references which can be exploited by our symbolization method. We summarize the patterns with the following assumption:

(A3) *d2c symbol references are only used as function pointers or jump table entries.*

By accepting A3, an n-byte value in data sections is lifted to a d2c reference if it is the starting address of some function, or it forms a jump table together with other n-byte values adjacent to it. Otherwise with A3 rejected, an n-byte data section value is symbolized whenever it is within the address space of code sections.

When A3 is taken, we will need to know whether a code section address is the start of a function. We also need to clarify what a jump table would be in the binary form. Identifying function beginnings in a binary is not a new research topic. Based on machine learning techniques, recent research [68] can reportedly identify function starting addresses with over 98% precision and recall. To avoid reinventing the wheel, we assume we have already known all the function start addresses. Since the binaries used in our research are all compiled from source code, we are able to get the ground truth by controlling the compilation and linking process.

Regarding the identification of jump tables, our algorithm is as follows:

- *Jump table start.* We traverse the data sections from the beginning to the end. If the address of an n-byte value is referred to by an instruction as the operand, it is considered as the first entry of a new jump table.

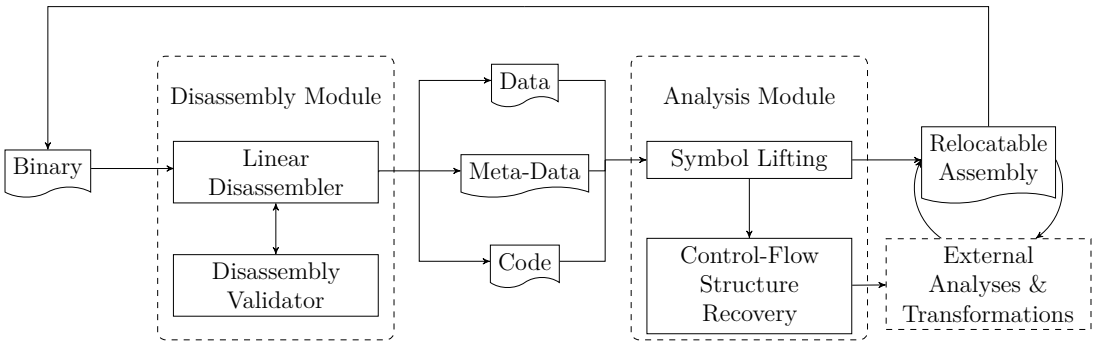


Figure 3.3: UROBOROS workflow.

- *Jump table entry.* If an n-byte value follows an already identified jump table entry, this value is also considered as an entry as long as it refers to instructions within the same function that previous entries point to.

The three assumptions A1, A2, and A3 are the basics of our symbolization method. With different choices of an assumption being applied or not, we can derive different strategies when processing a binary. Section 3.5 has a detailed evaluation on the correctness of reasonable combinations of these assumptions.

3.3 Design

3.3.1 Overview

The architecture of UROBOROS is shown in Figure 3.3. UROBOROS consists of two main modules—the disassembly module and the analysis module. The disassembly module decodes instructions with raw disassembling (Section 3.3.2) and dumps the data sections. The analysis module symbolizes memory references in both code and data sections (Section 3.2) and recovers the meta-information from the dumped content (Section 3.3.4). UROBOROS also recovers part of the control-flow structures from direct transfers so that it provides basic support for program-wide transformation (Section 3.3.3).

The disassembly module employs an interactive process to validate disassembled code from a linear disassembler. The linear disassembler decodes the code sections and dumps out all data and meta information sections. A validator is then invoked

to correct disassembly errors due to "data gaps" embedded inside code sections. The details are presented in Section 3.3.2.

After the raw disassembly is over, the dumped code, data, and meta-data are sent to the analysis module. This module identifies symbol references among immediate values in the code and data. As elaborated in Section 3.2, we propose three assumptions for reassembleable disassembling. The corresponding strategies are implemented in UROBOROS to guide the symbolization process. UROBOROS can be configured to utilize different combinations of assumptions for symbolization. We give a detailed evaluation on the correctness of different strategies in Section 3.5.1.

Given the symbolized instructions, the analysis module also partially recovers the control flows based on direct control-flow transfers. With the relocatable assembly and the basic control-flow structures, users of UROBOROS can easily perform advanced program analysis and program-wide transformations before they assemble the code back to binaries.

Finally, we emphasize that the assembly code generated and transformed by UROBOROS can be directly assembled back as a working binary by normal assemblers. In particular, the binary output is indeed a *normal* executable file without any abnormal characteristics such as patched or duplicated sections. Therefore, the reassembled binary can be disassembled again by UROBOROS or be processed by other reverse engineering tools.

We have implemented a prototype of UROBOROS in OCaml and Python, with a total of 13,209 lines of code. Our prototype works for both x86 and x64 ELF binaries.

3.3.2 Disassembly

In our prototypical implementation, the linear disassembler employed by UROBOROS's disassembly module is `objdump` from GNU Binutils. We implement an interactive disassembly process originally proposed in BinCFI [12].[2] In this process, the disassembler communicates with a validator which corrects disassembly errors due to "data gaps" between adjacent code blocks. The interactive procedure is as follows:

- `objdump` tries to decode the input binary for the first time.

[2]The BinCFI tool is available open source. We choose to reimplement the algorithm to make the codebase of UROBOROS more consistent such that it is fully automated and easy to extend. We refer readers to BinCFI [12] for the details of the disassembly process.

- The validator examines the output and check if there are explicit errors reported by `objdump`. In case there are no errors, the raw disassembly process terminates. Otherwise, the validator assumes the errors are caused by data embedded in code and computes the upper and lower bounds of identified "data gaps".
- With the computed range of identified "gaps", the validator guides `objdump` to decode the binary again, with those "gaps" skipped.
- Repeat this decode-validate process until no error occurs or the running time of the whole process reaches a time limit specified by users.

We leverage three rules proposed in BinCFI to validate the disassembly results and locate the data "gaps", i.e., "invalid opcode", "direct control transfers outside the current module", and "direct control transfer to the middle of an instruction". Since identifying bounds of each data gap can rely on the control-flow information of decoded instructions, the validator occasionally leverages UROBOROS's analysis module to retrieve the control-flow information.

3.3.3 Support for Program Transformation

UROBOROS provides basic support for program-wide transformations by partially recovering control-flow structures of the decoded instructions. We collect all the control transfer instructions to divide each function into multiple basic blocks. Control-flow graphs are rebuilt on top of these basic blocks. As a prototype, UROBOROS currently only processes direct control transfers. Regarding the intractable indirect transfers, a potential solution is to use value set analysis (VSA) [72] for destination computation. We leave including indirect control transfers in the CFG as future work.

3.3.4 Meta-Information Recovery

UROBOROS recovers the program-linkage table (PLT) and the export table in ELF binaries. The PLT table supports dynamic linkage by redirecting intra-module transfers on its stubs to external functions. As the base address of the PLT table can change after reassembling, we translate the memory references to PLT stubs to

their corresponding external function names, and let the linker to rebuild the PLT table with correct memory references during link time. In particular, this table is dumped out from the input binary and parsed into multiple entries, each containing the memory address of a PLT stub with its corresponding function name. Next, we scan the program and identify the addresses that match to a table entry. These addresses are then replaced by the corresponding function name.

Symbols need to be "exported" so that other compilation units can refer to them. The exported symbols together with their memory addresses are recorded in the export table. As ELF binaries do not keep a standalone export table, we construct this table by searching for all global objects in the symbol table. The symbol name of each entry and its memory address are then kept in a map. The export table can help identify functions and variables that are only referred to by other compilation units. We iterate each entry of the export table to insert symbols and `.globl` macros to the corresponding addresses.

For typical ELF binaries compiled from C code, `.eh_frame` and `.eh_frame_hdr` sections are used by compilers to store information for some rarely-used compiler-specific features, such as the "cleanup" attribute supported by GCC. For these sections, we dump the content out and directly write them back to the output. These sections are also used to store exception information for C++ programs.

3.3.5 Position Independent Code

Position independent code (PIC) typically employs a particular routine to obtain its memory address at run time. This address is then added by a fixed memory offset to access static data and code. According to our observation, the routine below is utilized by PIC code in 32-bit binaries to achieve relative addressing.

```
804C452: mov (%esp),%ebx
804C456: ret
```

PIC code invokes this routine by a `call` instruction, and register `ebx` is then assigned the value on top of the stack, which equals the return address. UROBOROS identifies this instruction pattern, traces the usage of `ebx`, and rewrites the instructions that add `ebx` with memory offsets to a relocatable format.

An example is shown in Figure 3.4. Once we identify a `call` instruction targeting the above sequence, we calculate the absolute address by adding `0x804c466` with

```
804c460: push %ebx
804c461: call 804c452
804c466: add $0x2b8e,%ebx
804c46c: sub $0x18,%esp

804c460: push %ebx
804c461: call S_0x804C452
804c466: add $_GLOBAL_OFFSET_TABLE_,%ebx
804c46c: sub $0x18,%esp
```

Figure 3.4: PIC code reuse.

offset `0x2b8e`, which equals `0x804eff4`. By querying the section information from ELF headers, `0x804eff4` equals the starting address of `.got.plt` table, and we rewrite offset `0x2b8e` to the corresponding symbol, which is `_GLOBAL_OFFSET_TABLE` in this case.

Theoretically PIC could use other patterns besides the above sequence to obtain its own memory address; the above instruction sequence is, however, the only PIC pattern we encountered after testing a broad range of real world applications (compiler and platform information is disclosed in Section 3.5).

As for x64 architectures, RIP-relative [73] memory references allow assembly code to access data and code relative to the current instruction by leveraging the `rip` register and memory offsets, which makes the implementation of PIC more flexible. In the raw disassembly output, instructions utilizing this mode are commented by `objdump` with the absolute addresses they refer to. We identify the comments, symbolize the memory offsets, and insert labels to the corresponding absolute addresses.

3.4 Optimization

3.4.1 Redundancy Trim

When a binary is dynamically linked to libc, the prologue and epilogue functions of the library are automatically added to the final product. UROBOROS attempts to support multiple iterations of the disassemble-reassemble process. Each time the binary is assembled, a new copy of the prologue and epilogue functions are inserted,

which unnecessarily expands binary size. Some tentative experiments show that binary size can grow 5 to 6 times larger with respect to the original, if we perform the disassemble-reassemble iteration for 1,000 times.

We cannot identify the prologue and epilogue functions in COTS binaries as the symbol information has been stripped. However, after the first disassemble-reassemble attempt, we get an unstripped binary with sufficient information indicating which functions are added by the linker. If we are to do another disassemble-reassemble round, UROBOROS can skip these functions in the disassembly phase.

Another source of redundancy is the padding bytes in data sections. In ELF binaries,there are three data sections (`.data`, `.rodata`, and `.bss`) that have padding bytes at the beginning. As these padding bytes are not used, we remove them from the recovered program before reassembling.

With the code and data redundancy trimmed, binary size expansion is reduced to almost zero, no matter how many times a binary is disassembled and reassembled.

3.4.2 Main Function Identification

In a compiler-produced object file, the symbol information of the `main` function is exported so that it can be accessed by the libc prologue functions in the linking process. However, as this symbol information in executable file is stripped in COTS binaries after linking, we need to recover and export it before reassembling.

Through our investigation, we found that the code sequence shown below is typically used to pass the starting address of `main` to libc prologue function `libc_start_main`.

```
push $0x80483b4
call 80482f0 <__libc_start_main@plt>
hlt
```

The first argument of `libc_start_main`, which is `0x80483b4` in this example, is recognized as the starting address of the `main` function. We insert a label named `main` and the type macro `.globl main` in the output at this address.

3.4.3 Interface to External Transformation

In order to demonstrate that UROBOROS is an enabling tool that makes analysis and transformations applicable to legacy binaries in general, we implement a

Table 3.1: Programs used in UROBOROS evaluation.

Collection	Size	Content
COREUTILS	103	GNU Core Utilities
REAL	7	bc, ctags, gzip, mongoose, nweb, oftpd, thttpd
SPEC	12	C programs in SPEC2006

diversification transformation based on basic block reordering. After disassembly, we walk through each function and randomly select two basic blocks from its CFG as the reordering targets. Control-flow transfer instructions and labels are inserted in the selected blocks, their predecessors, and successors to guarantee semantic equivalence. We perform this reordering *iteratively*, namely the output of each iteration becomes the input of the next round. We conducted a quick experiment on `gzip`. The disassembly-transformation-reassembly process was iterated 1,000 times. The effectiveness of the diversification transformation is evaluated by the elimination rate of ROP gadgets measured by the ROP gadget detector ROPGadget [74]. From this preliminary study, we find that it is much easier than binary rewriting to perform binary-based software retrofitting based on UROBOROS. As the ROP defense is not the focus of this research, we omit the detailed results in this work.

3.5 Evaluation

We evaluate UROBOROS with respect to correctness, cost, and its ability to support program-wide transformation. The correctness verification examines whether UROBOROS's reassembly is semantic preserving. Evaluation on the cost of UROBOROS reveals its reassembly's impact on binary size and execution speed, and also the running time of UROBOROS itself. As presented in Section 3.4.3, we study UROBOROS's support for binary-based software retrofitting, by implementing a basic block reordering algorithm to diversify disassembled binaries and eliminate ROP gadgets. As we have emphasized, UROBOROS is an enabling tool for other security hardening techniques. However, as goal-driven software security hardening is out of the scope of this work, we do not present the detailed experiment results here.

Table 3.2: Functionality test input for Real.

Program	Test Input
bc	Test cases shipped with the program
gzip	Test cases shipped with the program
ctags	Parse a C source file of 152,270 lines
oftpd	Login and fetch a large file
thttpd	Request some web pages & a large file
mongoose	Request some web pages & a large file
nweb	Request some web pages & a large file

We use three collections of binaries compiled from C code to evaluate Uroboros. The first set, referred to as Coreutils, is the entire GNU core utilities including 103 utility programs for file, shell, and text manipulation. The second set, called Real, consists of 7 real-world programs picked by us, covering multiple categories such as floating-point and network programs. The last set subsumes all the C programs in the SPEC2006 benchmark suit, thus will be denoted by Spec. Details of each collection are listed in Table 3.1. In the evaluation we compile all programs for both 32-bit and 64-bit targets. Since there are 122 programs, the number of tested binaries is 244 in total. The compiler is GCC 4.6.3, using the default configuration and optimization level of each program. All experiments are undertaken on Ubuntu 12.04. For each test case, we use the `strip` tool from GNU Binutils to strip off the symbol information and debug information before testing.

3.5.1 Correctness

We verify the correctness of Uroboros's reassembility in two steps. First, we execute binaries assembled from Uroboros's output with test input shipped with the software. Both Coreutils and Spec have test cases shipped with the software by default. As for the Real programs, most of them do not have test cases, so we develop input by ourselves to verify the major functionality. The input we use for testing the Real collection is listed in Table 3.2.

Second, we examine the false positives and false negatives of our symbolization process for all the binaries of the three collections. In our context, a false positive is an immediate value that we mistakenly symbolize, while a false negative is a symbol reference that we fail to identify.

Table 3.3: Dynamic test results on reassembled binaries.

Assumption Set	Binaries Failing Functionality Tests	
	32-bit	64-bit
{}	h264ref, gcc, gobmk, hmmer	perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lbm, sphinx3
{A1}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A2}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A3}	gobmk	gcc, gobmk
{A1, A2, A3}	gobmk	

As described in Section 3.2, we have different assumptions to guide the symbolization process, so the correctness of different assumption combinations are verified. Since the three assumptions are orthogonal, there are eight different combinations with the choices of the three assumptions. With limited resources, it is difficult to test all 244 programs on all assumption sets. With some tentative experiments, we found that A1 is an assumption which greatly improves the overall performance of our disassembly and reassembly method. Therefore, we reduce the eight candidates to five by always including A1 except in the empty assumption set. In detail, the five assumption sets applied are {} (empty set), {A1}, {A1, A2}, {A1, A3}, and {A1, A2, A3}.

For all tested assumption sets, all reassembled binaries from COREUTILS and REAL pass the functionality tests. Some binaries from SPEC, however, fail to pass the tests, which are listed in Table 3.3. With the assumption set {A1, A2, A3}, only the 32-bit version of `gobmk` from SPEC (out of 244 cases in total) fails the functionality test. By inspecting this defected binary, we successfully locate the cause of failure. Some 4-byte sequences in the data sections happen to contain the same value as the starting address of a function, but they are not code pointers. UROBOROS incorrectly symbolizes them, leading to false positives. After we correct these errors, `gobmk` successfully passes the test.

For symbol-level correctness verification, we provide the statistics on false positives and false negatives of symbolization. A false positive is an immediate value that should not have been symbolized. A false negative is an immediate value which should be symbolized but failed to be after our symbolization process. We obtain the ground truth by parsing the relocation information provided by the linker.

We have verified all binaries in this step. We now list the results for non-trivial cases, namely programs with at least one symbolization false positive or false

Table 3.4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive).

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	1	0.049‰	1	0.049‰	0	0.000‰	0	0.000‰
gcc	262698	49	0.187‰	32	0.122‰	32	0.122‰	0	0.000‰	0	0.000‰
gobmk	65244	1348	20.661‰	985	15.097‰	912	13.978‰	78	1.196‰	5	0.077‰

Table 3.5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative).

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gcc	262698	11	0.042‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gobmk	65244	86	1.318‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

negative with any assumption combination. Table 3.4 and 3.5 show the false positive and false negative analysis for 32-bit binaries, and Table 3.6 reports false positive analysis for 64-bit binaries. There are no false negatives on any of the 64-bit binaries. We emphasize in particular that, *with {A1, A2, A3} applied, among all the 244 binaries, only* `gobmk` *has a few false positives, and none has false negatives.*

The results of symbol-level verification are highly synchronized with the results from the first stage—binaries reassembled with no false positives or false negatives can pass all test cases. The results show that symbolization errors are found in `gobmk` no matter which assumption set we apply. In particular, we have verified that symbolization errors found in `gobmk` when applying {A1, A2, A3} are all caused by program data colliding with some function starting addresses. These collisions cause a functionality test failure for 32-bit `gobmk`, but the 64-bit version can pass the test due to the incompleteness of test input. In summary, the two stages of verification together imply that all three assumptions proposed for symbolization are reasonable.

Although the symbolization errors occurring in the case of `gobmk` seem conceptually "general", our study shows that the collisions are actually rare in practice, unless the disassembled binary has very large data sections like `gobmk` does. On the other hand, UROBOROS can successfully disassemble large and complicated

Table 3.6: Symbolization false positives of 64-bit SPEC, REAL and COREUTILS (Others have zero false positive). Also, no false negatives are found for any binary.

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76952	32	0.416‰	10	0.130‰	10	0.130‰	0	0.000‰	0	0.000‰
gcc	259213	506	1.952‰	126	0.486‰	14	0.054‰	112	0.432‰	0	0.000‰
gobmk	65255	2437	37.346‰	1079	16.535‰	7	0.107‰	1073	16.443‰	1	0.015‰
hmmer	13165	11	0.836‰	2	0.152‰	0	0.000‰	2	0.152‰	0	0.000‰
sjeng	8837	22	2.490‰	2	0.226‰	0	0.000‰	2	0.226‰	0	0.000‰
h264ref	20264	15	0.740‰	1	0.049‰	0	0.000‰	1	0.049‰	0	0.000‰
lbm	248	1	4.032‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
sphinx3	8656	3	0.347‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
ctags	12997	2	0.154‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gzip	3323	11	3.310‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
mongoose	3643	1	0.275‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
df	4202	1	0.238‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
du	4593	1	0.218‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
split	2851	1	0.351‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
timeout	1935	1	0.517‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

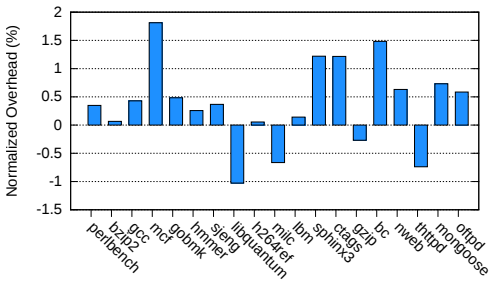


Figure 3.5: Execution overhead for REAL and SPEC programs relative to the original versions.

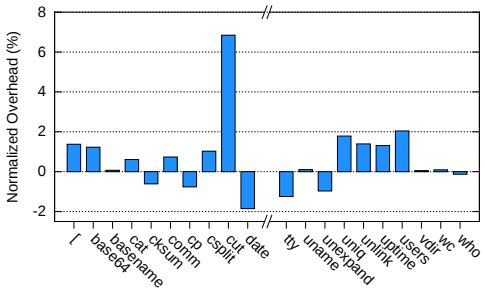


Figure 3.6: Execution overhead for COREUTILS programs relative to the original versions.

binaries like `gcc` and `perlbench`. Overall, the results from two stages of correctness verification suggest that UROBOROS is a promising tool with remarkable practical value.

3.5.2 Cost

The cost of UROBOROS manifests from three aspects: size expansion of reassembled binaries, execution overhead of reassembled binaries, and the processing time of UROBOROS itself.We report the evaluation results on 32-bit binaries in this paper.

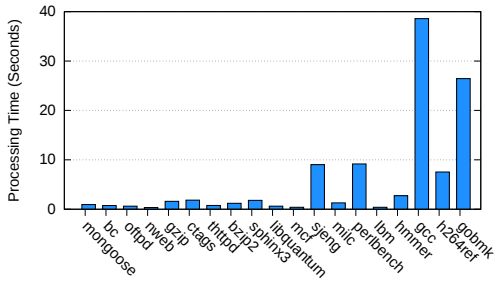


Figure 3.7: Processing time for SPEC and REAL binaries.

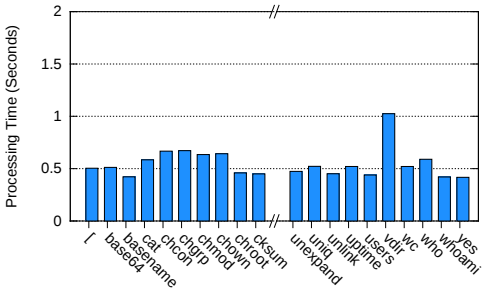


Figure 3.8: Processing time for COREUTILS binaries.

3.5.2.1 Execution Overhead

Some programs in COREUTILS are not suitable for performance benchmarking, including `su`, `nohup`, and `timeout`, etc. After excluding these programs, we have 90 left to inspect in COREUTILS. The experiments are conducted on a machine with Intel Core i7-3770 3.40GHz and 8GB memory running Ubuntu 12.04.

We present the execution slowdown of reassembled binaries in Figure 3.5 and Figure 3.6. Since it is hard to present the data of all 90 binaries from COREUTILS, we sort COREUTILS programs by their names in alphabet order and plot the data for the first and last 10 programs in Figure 3.6. We report that the average slowdown for is 0.44% for COREUTILS, 0.29% for SPEC and 0.52% for REAL. The data suggests that UROBOROS does not have any significant impact on the execution speed of reassembled binaries.

3.5.2.2 Size Expansion

We use the `stat` program from GNU Coreutils to calculate file size expansion of the reassembled binaries compared to the originals. As the increase is generally negligible, we only report the average data here. The average expansion for COREUTILS is 0.83%, 0.00% for SPEC and -0.02% for REAL. Data shows that UROBOROS has almost zero impact on binary size when delivering reassembility. As aforementioned in Section 3.4.1, subsequent disassembly-reassembly iterations have zero expansion.

3.5.2.3 Processing Time

We measure how long it takes UROBOROS to disassemble binaries. Figure 3.7 presents the processing time for SPEC and REAL binaries. Figure 3.8 presents processing time for COREUTILS binaries selected using a same alphabet order strategy. As expected, larger binaries take more time to process. On average, UROBOROS spends 8.27 seconds on binaries from SPEC, 0.98 seconds on binaries from REAL, and 0.57 seconds on binaries from COREUTILS. We interpret this as a promising result, and the efficiency of UROBOROS makes it a tool totally practical for production deployment.

Chapter 4
Instrumentation Framework

Software instrumentation techniques are widely used in program analysis tasks such as program profiling, vulnerability discovering, and security-oriented transforming. Enabled by our novel disassembling technique, in this chapter we present a generic binary instrumentation platform called UROBOROS. UROBOROS overcomes the limitation of existing static instrumentation tools and supports low-cost instrumentation on stripped binaries. In this chapter, we first give a platform overview in Section 4.1 and elaborate on the instrumentation capability of UROBOROS in Section 4.2. We then present the platform design in Section 4.3. We evaluate the UROBOROS instrumentation cost in Section 4.4, and present two sample applications in Section 4.5.[1]

4.1 Platform Overview

Software instrumentation inserts extra instructions to the target program to achieve multifaceted tasks. For example, instrumented code can record process run-time behavior [41], support program analysis [14, 45], or harden the executable layout to improve security [12, 13]. The instrumentation task can be performed at different stages: at compile time [1], at run time [14], or statically on executable files [12, 13]. We present UROBOROS, a tool performing static instrumentation on stripped binaries.

The primary target of UROBOROS is stripped binaries, i.e., binaries with no debug or relocation information. In order to hinder reverse engineering and reduce

[1]The work of this chapter is published in the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering [75].

the executable size, debug and relocation information is usually removed from binaries before releasing to the public. Without sufficient program information, reverse engineering on stripped binaries could be problematic and fragmentary. In particular, as the disassembled output of stripped binaries is *unrelocatable*, program transformations have to be very conservative for correctness, leading to enormous challenges for binary instrumentation. With several novel disassembling methods [40], UROBOROS recovers relocatability for stripped binaries. This enables painless program instrumentation because on relocatable assembly programs users can statically inline extra code to the instrumentation points. The instrumentation output, including both the original program and the instrumentation code, can be then assembled back into normal binaries. In sum, UROBOROS delivers complete, ease-to-use, efficient, and transparent instrumentation on stripped binaries.

Complete. Most existing work leverages dynamic analysis to recover relocation information, in which only incomplete functional components are obtained from the input [30, 31]. UROBOROS disassembles input binaries and recovers relocation information in both data and code sections through advanced static analysis. As the whole disassembled output is *relocatable*, program-wide transformations become feasible without common binary instrumentation drawbacks introduced by unrelocatable code or data snippets.

Easy-to-use. Existing static binary instrumentation tools need to rewrite the input binary. The instrumentation code is usually attached to the rewritten output to generate a workable executable [24, 37, 46, 47]. Although the rewriting process on stripped binaries could be complicated, tedious, and even problematic, the static instrumentation facilities of UROBOROS are make it easy, even for users with *no* binary rewriting skills. In fact, distinguished from existing static binary instrumentation approaches, UROBOROS does not need to rewrite the input binary. Legacy binaries are disassembled into *relocatable* programs, which can be instrumented as easily as compiler-generated assembly code. Users with *only* source code analysis and transformation skills can find no difficulty in using UROBOROS because they are essentially facing the same tasks they are familiar with.

Transparent. UROBOROS performs *reassembly-based instrumentation*. Previous static instrumentation tools rewrite the input binary to patch the instrumentation code or replicate the original code section into two and instrument both. In contrast, instrumentation code of UROBOROS is directly inlined into each instrumentation

point. Transparent instrumentation enables UROBOROS itself or any other binary instrumentation tool to re-process the UROBOROS output easily. This feature can greatly broaden the application scope by bridging UROBOROS with existing infrastructures and potentially puts UROBOROS as the foundation of most binary analysis tasks.

Efficient. As jump instructions are frequently used to redirect control transfers between original code sections and patched sections, existing static binary instrumentation tools can incur relatively high execution slowdown and size increase on the rewritten output. However, as UROBOROS inlines instrumentation code at each instrumentation point, instrumentation code is connected with the context by "fall-through" transfers. In fact as jump instructions are not used anymore, UROBOROS engenders no additional execution cost from the instrumentation process, which is very efficient compared with previous binary instrumentation tools.

4.1.1 Platform Highlights

Different from the existing binary instrumentation techniques, UROBOROS delivers static reassembly-based instrumentation, i.e., the instrumentation output can be readily reassembled back to generate a binary again. UROBOROS recovers relocatable assembly code, enabling painless program-wide transformations. Due to the lack of relocation information, traditional static binary instrumentation (SBI) tools insert jump instructions at instrumentation targets to redirect control flow transfers. Note that the inserted jump instructions can bring in a non-negligible performance penalty in the instrumentation outputs. Furthermore, dynamic binary instrumentation (DBI) tools hook the target process and instrument the program during run time, which can lead to even higher execution cost. By contrast, benefiting from relocatability, no additional overhead is introduced by UROBOROS, which is very efficient compared with the existing SBI and DBI tools.

4.1.1.1 An Easy-To-Use Rich Instrumentation API

UROBOROS translates the input binary into its internal representation and then recovers the program control flow structures on top of the representation. It provides an easy-to-use rich API to inspect and manipulate the internal representation and program structures. Currently, UROBOROS provides access to data bytes,

instructions, basic blocks, functions, control flow graphs (CFGs) and call graphs (CGs). We demonstrate the versatility of the UROBOROS API by instrumenting input binaries at different levels of the program structure and also presenting two real-world instrumentation applications.

4.1.1.2 Enable Novel Applications and Boost Existing Applications

We present and evaluate two applications on top of UROBOROS. To our best knowledge, UROBOROS enables iterative software diversification on *stripped binaries*. On trace profiling, UROBOROS delivers significantly better instrumentation performance. We present an in-depth study of UROBOROS comparing with the industry-standard DBI tool, Pin [41]. We summarize our results as follows:

- *Iterative diversification.* A novel disassemble-diversify-reassemble workflow is enabled by UROBOROS. The diversified output is reprocessed for multiple times, boosting the diversification due to the "iteration effect". We observed notable binary similarity score decreasing with more iterations of processing.
- *Trace profiling on SPEC2006 and Linux common utilities.* UROBOROS incurs around 2.37X execution slowdown comparing with the native execution, while Pin imposes a sharp 8.83X slowdown on average.

4.2 Static Binary Instrumentation

The UROBOROS API enables static instrumentation on stripped binaries in a program-wide scope. In this section, we first compare reassembly-based binary instrumentation with two commonly-used static binary instrumentation strategies used in the previous tools, i.e., patch-based instrumentation and replica-based instrumentation. We then demonstrate the UROBOROS instrumentation capability by creating an instrumentation application to trace memory writes.

4.2.1 Existing Binary Instrumentation Techniques

As previous binary disassemblers only recover *unrelocatable* assembly programs, existing static instrumentation tools have to deliberately instrument input binaries without breaking memory references. To rebuild the instrumentation code and the

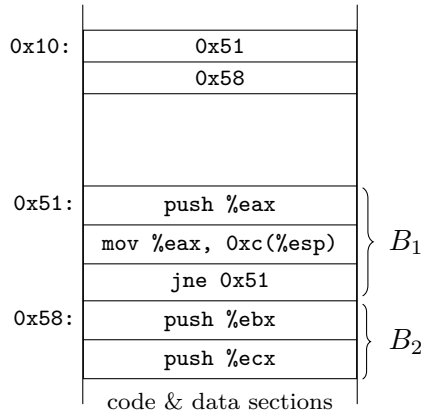


Figure 4.1: An example of instrumentation target.

input binary into a workable output, two rewriting methods are usually used, i.e., patch-based instrumentation and replica-based instrumentation. Instrumentation tools using the first strategy patch instrumentation code as a new section to the input binary. Jump instructions are inserted in the original code which redirect control transfers to the patched section. Replica-based instrumentation duplicates the original code sections into two; the replica is instrumented while jump instructions are inserted to the original which forward indirect control transfers to the replica. Both existing strategies could become challenging, and it may require users with specific rewriting skills to handle the whole process. In this section, we take a binary instrumentation task as an example to compare reassembly-based instrumentation with existing methods. Figure 4.1 presents the layout of a stripped binary. Note that memory references (e.g., 0x51) are all *unrelocatable* immediate values. The code section contains two basic blocks (B_1, B_2); suppose we want to instrument basic block B_1 to add a counter instruction at the beginning.

4.2.1.1 Patch-Based Instrumentation

Patch-based instrumentation replaces instructions at the instrumentation point with a jump instruction, which points to a new section patched at the end of the input binary. The newly added section contains both the instrumentation code and the replaced instructions. Figure 4.2 presents the code layout after patching. As a long jump (e.g., "`jmp Patch`") needs to occupy 5 bytes, two instructions are relocated to leave enough space. During run time, the inserted jump instruction redirects the control flow to the patched section, and after executing the instructions on the

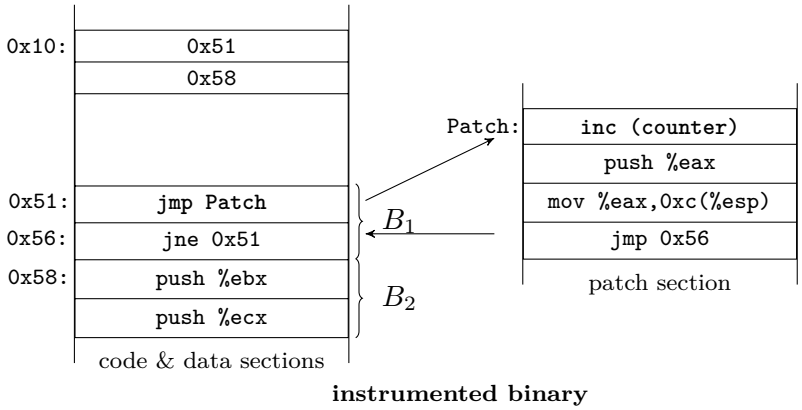


Figure 4.2: Patch-based instrumentation.

patched section, another jump instruction redirects control flow back to the original code section. As two control transfers could occur at each instrumentation point, this rewriting strategy introduces relatively high performance slowdown. Even more seriously, usually more than one instruction have to be replaced, and the replaced instructions are required to be *relocatable* in order to keep the correct semantics. It is not always obvious whether these instructions can be safely relocated. Optimization techniques are proposed to use a short jump or even an interrupt (`int 3`) to transfer control to the patched section, but short jump (2 bytes on x86 architecture) still cannot substitute a single byte instruction and frequent interrupt handlings have a big penalty on the execution [38, 76].

4.2.1.2 Replica-Based Instrumentation

Typical instrumentation tools in this category generate a replica of the original code section. The replica contains both the instrumentation and the original code. As the replica has a different range of memory addresses, memory references in the original code, e.g., destinations of function calls, are cautiously rewritten with new memory addresses in the replica at the best effort. Jump instructions are deliberately inserted at control flow destinations in the original copy, forwarding indirect memory references to their new targets in the replica in case some address translations are missed and the execution control is transferred to the original copy. This replica-based instrumentation can reduce performance slowdown caused by frequent control transfers in the instrumentation output, especially when there are

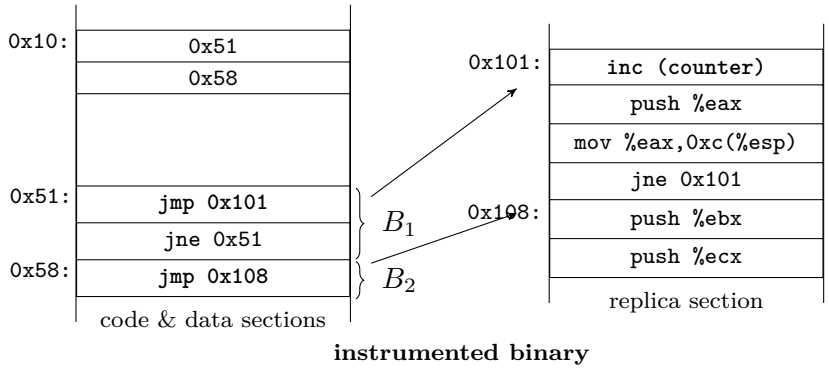


Figure 4.3: Replica-based instrumentation.

large amounts of instrumentation targets, e.g., all the basic blocks. However, the size of rewritten output can be notably increased.

In general, the rewritten output by either patch-based or replica-based instrumentation changes the internal structure of input binaries, and it becomes very challenging or even impossible to apply binary analysis and transformation on the instrumented binaries. Our experiments in Section 4.4 report that disassembling the rewritten outputs can lead to many decoding errors, while no error is found when disassembling their original inputs. Deng, Zhang, and Xu [77] also discuss how patch-based and replica-based instrumentations can impede binary component extraction and embedding. Overall, in addition to high cost imposed on the instrumentation output, existing instrumentation tools cannot undertake *transparent* instrumentation, which narrows their application scope as well.

4.2.2 Reassembly-Based Instrumentation

As UROBOROS can recover an executable in a *relocatable* format before instrumentation, the instrumentation code can be directly *inlined* into the target. The instrumentation output is then assembled to produce a *normal* binary output. As shown in Figure 4.4, memory references in the unrelocatable program have been translated into relocatable formats (e.g., "`S_0x01`" and "`S_0x51`"), and UROBOROS directly inserts the counter instruction at the instrumentation point. As the counter instruction is inlined in the context, instrumentation cost introduced by frequent control transfers or replicated code is indeed avoided. Moreover, given all the memory references in a relocatable format, linkers will resolve these references with

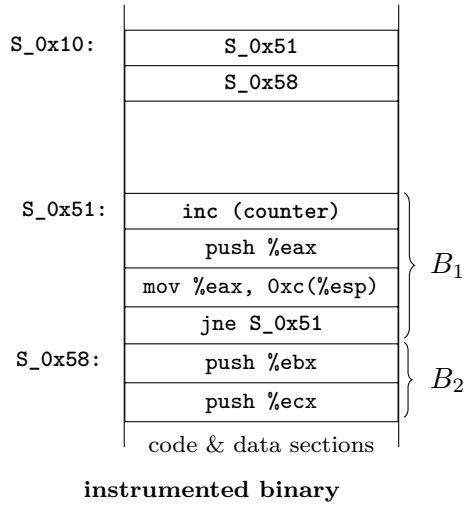


Figure 4.4: Reassembly-based instrumentation.

new memory addresses during reassembling. Code pointers can naturally refer to their original destinations at this time, and no intentional binary rewriting is needed to adjust the value of code pointers. In addition, the instrumentation output can be seamlessly reprocessed by further binary analysis or transformation without any particular difficulty introduced by the UROBOROS instrumentation.

4.2.3 Instrumentation Samples

In Figure 4.5, we present the code that a user needs to write if he wants to trace memory writes. The code is written in the OCaml programming language. The `process` function (line 20) utilizes the UROBOROS `Instr_visitor` module to traverse the whole instruction list and iterate all the memory writing instructions. By leveraging the `is_mem_write` function (line 23) from the `Instr_utils` module, memory writing instructions are filtered out during iterating. Users only need to define their `visit` function (line 22), and UROBOROS takes care of the underlying details. Memory writing instructions are classified into three categories according to the number of operands each instruction has (line 8). The `Instr_template` module provides the `gen_logging_instrs` function (line 14) to generate logging instructions, which initializes a sequence of instructions for logging according to the target instruction and its location information. Finally, the `insert_instr_list` function (line 24) from the `Instr_utils` module updates programs by inserting instrumen-

```
open Type
open Utils
open Instr_utils
let il_update = ref []

let instrument i t =
  let module IT = Instr_template in
  match t with
    | SINGLE_WRITE
    | DOUBLE_WRITE
    | TRIPLE_WRITE ->
      il_update :=
      (get_loc i
      |> IT.gen_logging_instrs i) @ !il_update;
    (* relocation labels on instrumentation target have
     * been given to inserted code; remove redundancy *)
      eliminate_label i
    |_ -> i

let process il =
  let module IV = Instr_visitor in
  let visit i t = instrument i t in
  IV.map_instr is_mem_write visit il
  |> insert_instr_list BEFORE !il_update
```

Figure 4.5: A UROBOROS plugin for tracing memory writes.

tation code at instrumentation points. This function supports instrumentation before or after target instructions.

By providing sufficient instrumentation facilities, UROBOROS only exposes high-level interfaces to users. In the above example, we visit programs at the instruction level and search for memory write operations. The `Instr_utils` module provides various instruction filtering utilities (e.g., `is_mem_write`, `is_jmp`) to help users classify instructions according to the functionality. UROBOROS also provides a rich API to traverse and access multiple levels of the program structure. More examples are presented in Section 4.5.

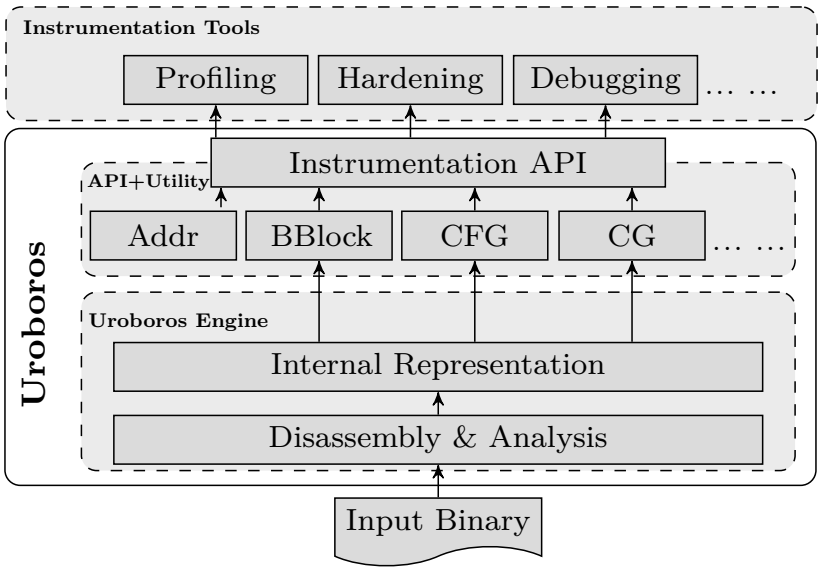


Figure 4.6: The architecture of UROBOROS.

4.3 Design

4.3.1 Platform Structure

The design overview is shown in Figure 5.2. UROBOROS consists of three main modules—the disassembly module, the analysis module, and the instrumentation module. The disassembly module takes executable files as inputs and recovers unrelocatable disassembled outputs. The disassembly module also dumps the data and metadata sections from the input. The analysis module identifies memory addresses from all the immediate values found in the input and lifts them into relocatable symbols. The analysis module also recovers control flow structures on top of the relocatable program. The recovered program is parsed into multiple levels of internal representations, and the instrumentation module provides utilities to access and manipulate these internal data structures. Users can utilize the provided facilities to undertake program-wide analysis and transformations on the relocatable program. As we have emphasized, the instrumented output of UROBOROS can be directly assembled back into a normal binary, which can even be re-processed by UROBOROS for multiple times.

The disassembly module of UROBOROS implements the disassembling algorithm proposed in BinCFI [12]. We consider data bytes embedded in code sections the main

reason to make linear disassembling fail. In this algorithm, the linear disassembler leverages a validator to correct disassembling errors due to the embedded data bytes. As suggested by BinCFI, we rely on three rules to validate the raw disassembling results and locate "data gaps", i.e., invalid opcode, direct control transfers outside the current module, and direct control transfer to the middle of an instruction. According to these rules, we implement a multiple-round disassembling validation process.

Various kinds of relocatable symbols exist in compiler generated assembly programs, e.g., jump table entries, function pointers. During the link time, linker substitutes each relocatable symbol with a concrete value, which represents the runtime memory address of the corresponding object . To make the disassembled output relocatable, the main challenge is to identify memory addresses from all the data found in the disassembled output. The identified memory addresses can be then translated into relocatable symbols that enable the reassembly-based instrumentation. However, it is indeed quite challenging to find memory addresses in the disassembled output, as theoretically it is impossible to decide statically any suspicious immediate value as a memory address or some random data without execution. We leverage multiple methods to guide UROBOROS in identifying memory addresses from immediate values. These methods are proposed based on observations of the disassembled outputs from real-world programs [40]. An in-depth study shows that with all the methods applied, almost no false positive or negative is found on broad sets of real world binaries.

4.3.2 Instrumentation Module

4.3.2.1 Internal Data Representation

UROBOROS supports program-wide instrumentation by recovering control flow structures of the input program. The disassembled instructions, dumped data sections, and metadata are parsed into an intermediate representation for analysis and transformations. We now introduce how we organize these internal representations.

We have created a hierarchical representation for the recovered program to support program-wide analysis and transformation. The topmost level in this hierarchy is the *Module* object, which corresponds to each disassembled binary. This object stores the binary-level information of inputs. Typically program binary stores

its data, code, and metadata in different binary sections, and UROBOROS keeps each section in one object. Control structure units, i.e., basic blocks and functions, are stored in their corresponding *BBlock* and *Function* objects. UROBOROS also maintains *Instruction* objects, which represent instructions in the code section. Three data sections (`.rodata`, `.data`, and `.bss`) are stored inside UROBOROS, and UROBOROS uses one object to represent each data byte. Metadata from the input binaries is stored in the same way. Furthermore, each memory address is maintained in an `Addr` object. `Addr` objects maintain both memory labels and memory addresses. All the internal modules have their associated `Addr` objects. Labels and memory addresses can be updated to support arbitrary manipulations that may change memory addresses in the transformed output.

4.3.2.2 Support Binary Instrumentation

Stripped binaries usually do not contain sufficient control flow information, e.g., functions, basic blocks, or control flow graphs. This information is recovered in the UROBOROS analysis module. UROBOROS recovers function entry point addresses by collecting destinations in `call` instructions and exported function information from the symbol table.[2] UROBOROS also identifies functions by matching instructions with typical function starting patterns; 52 instruction patterns are implemented in this step. In addition, UROBOROS can be configured with user provided function entry point addresses.

UROBOROS recovers basic blocks as the units of the control flow structure. Basic blocks are identified in its standard way, i.e., instruction sequences with only one control flow entry point and one exit point. We collect all the identified code pointers and control flow instructions to divide instructions into multiple blocks. Note that as the destinations of most indirect jump instructions are code pointers that can be found in code and data sections, UROBOROS is capable of identifying basic blocks determined by indirect jumps according to the collected pointers. Each basic block object maintains information of its predecessors and successors on the CFG. For indirect control flow transfers, we conservatively consider they can transfer to the beginning of any basic block.

[2]Note that a symbol table does often exist in stripped binaries that contains, for example, "exported function" information to support dynamic linkage.

UROBOROS and other static instrumentation tools face similar challenges (undecidability issues), for example, basic block and function recognition [68, 69]. We are currently working on more sophisticated strategies to recover function boundaries, and these techniques will be merged into the UROBOROS future releases. Besides, the identification of indirect control transfer destinations can be improved through value-set analysis (VSA) [72], which is left for our future work.

In general, 21 modules, including 121 functions, are implemented in UROBOROS to support binary instrumentation. These utilities include common operations at different levels of the program structure (e.g., function, basic block and instruction). For instance, UROBOROS provides multiple visitor modules to support visiting different levels of the program structure in a flexible way, e.g., `Function_visitor`, `BBlock_visitor`, and `Instr_visitor`. Utility modules are also provided to query, modify, and remove internal objects. For example, UROBOROS provides `Instr_utils` and `BB_utils` modules to support inspection and manipulation on `Instruction` and `BBlock` objects. In addition, the `Instr_utils` module provides functions to query instruction types, for example, `is_mem_write`, `is_mem_read`, `is_call`, `get_label`, and `get_addr`. If modules (*Instruction*, for instance) are maintained in a list, UROBOROS also provides utilities to traverse all the entries on the list one by one.

4.3.2.3 Instrumentation on Assembly Code

Currently UROBOROS supports inserting instrumentation instructions in the format of its internal representation. Instrumentation code can be inserted as follows:

```
insert_single_instr pos loc i_insert il
```

This function inserts instrumentation instructions at the given location. `Pos` is a predefined type, including both `BEFORE` and `AFTER` type variables. Instrumentation code can be inlined before or after the target instruction. `BEFORE` means inserting instrumentation code in front of the target, while `AFTER` means behind. `i_insert` represents the assembly instructions in the UROBOROS internal data structures. `loc` consists both memory address and possible relocation labels referring to that address. `il` is the internal representation of input program instructions.

As shown in Figure 4.5, another frequently-used function is `insert_instr_list`. In case a large amount of instrumentation code needs to be inserted at different

places, users need to construct each instrumentation instruction with its desired instrumentation position. The instrumentation instruction list is then provided to this function, as well as the `pos` type. This function sorts the instrumentation code according to their desired memory positions and inlines instrumentation instructions into the targets.

Besides instrumentation code insertion, we also provide functions to replace existing instructions with instrumentation code as follows:

```
sub_single_instr i_t i_sub il
```

This function uses instrumentation code to replace the given target. Here, `i_t` and `i_sub` represent the target instruction and its substitution while `il` stands for the whole instruction list. UROBOROS also provides function `instrument_update` to support instrumentation code updating in both *insertion* and *substitution* operations. This function requires users to construct bundles which consist of instrumentation code and the associated instrumentation types, i.e., `INSERT` or `SUB`.

In addition, UROBOROS can output the text representation of assembly code which can further facilitate ad-hoc or customized analysis and transformation.

4.3.2.4 CPU Flags Usage Optimization

Many instrumentation scenarios require the instrumentation code to have a counter; whenever execution flow hits the instrumentation point, the counter is incremented by a fixed stride. The maintained counter can be used to record the number of executed instructions, or as the index of an array to record some execution information. However, opcodes (`inc`, for instance) that are usually used to increment the counter need to change CPU flags, and in order to preserve the correct semantics, CPU flags need to be saved and restored for instrumentation. Opcodes `pushf` and `popf` are designed to save and restore CPU flags on the stack. However, both opcodes can delay the instruction pipelining and cause relatively high performance slowdown.[3] In fact, some of our early experiments show that performance slowdown could reach almost 700% for a trace profiling instrumentation on `gzip`.

Instead, UROBOROS deliberately selects instructions to optimize the flag manipulation operations. Templates are provided to users so that they can easily construct optimized instrumentation code. For the commonly used counter-increment sce-

[3]The instruction `pushf` takes 3 uops while `popf` takes 9 uops on the Intel Haswell architecture.

nario, UROBOROS can even avoid changing CPU flags by misusing the `loop` opcode.[4] `Loop` does not change any flag and decreases the value in register `ecx` each time until it reaches zero. A classic example is shown below, which records the number of executed basic blocks.

```
BB:       push %ecx
          movl index, %ecx
          loop BB_stub
BB_stub:  movl %ecx, index
          pop  %ecx
```

On 32-bit x86 architecture, the global variable `index` needs to be initialized with `0xffffffff` before execution, and the total number of executed basic blocks can be calculated by subtracting `0xffffffff` with the final value in `index`.

Another optimization is to use opcode `lahf` and `sahf` to speed up the saving and restoring of CPU flags. However, tests show that `lahf` and `sahf` slow down the execution for about 15% compared with the "loop" optimization. Also these two opcodes are absent for early AMD and Intel CPUs. UROBOROS still provides templates using these opcodes to construct instrumentation code for the sake of handling more general cases. The accessibility of low-level details makes UROBOROS quite flexible when facing some cost-sensitive instrumentation scenarios.

4.4 Evaluation

UROBOROS instrumentation is directly applied to the relocatable assembly, and only negligible cost is introduced without actually inserting instrumentation code. The experiments have shown a trivial instrumentation, i.e., disassembling a binary and reassembling it as what it was, leads to at most 1% execution slowdown and size expansion, when evaluating a broad set of real world program binaries.

We do not report the detailed results of this trivial evaluation, for the data is not very informative. Instead, we compare UROBOROS with another static instrumentation tool DynInst [46, 47] with respect to the performance and size of instrumented binaries. We also evaluate the execution time of UROBOROS itself to

[4] We learned this optimization from a discussion at `http://goo.gl/FbODjk`. The trick was initially suggested by Guntram Blohm.

demonstrate its efficiency. We set up all the experiments on a server machine with a 2.90GHz Xeon(R) E5-2690 CPU and 128GB RAM.

We pick DynInst (version 8.2.1, `http://www.dyninst.org/`), a widely used static binary instrumentation framework, as the competitor of UROBOROS. DynInst features both patch-based and replica-based instrumentation. For instrumentation tasks with small amount of instrumentation targets, DynInst undertakes a patch-based instrumentation, in which two control transfers occur before and after executing instrumentation code. However, patch-based instrumentation could result in many additional control transfers when instrumenting large amount of targets (e.g., instrumentations on every basic block or function). In that case, DynInst will switch to the replica-based instrumentation to reduce the execution overhead at the expense of code size bloating.

We use basic block and function counting instrumentation as the benchmark to evaluate both tools on all the C programs from 32-bit SPEC2006. We record execution slowdown and size increase for the instrumented binaries. We compile all the test cases from source code with the default configurations. All the test cases are stripped before processing by UROBOROS (with the `strip` tool from GNU Binutils).[5] Before the cost evaluation, we first verify the functionality of the instrumentation output with test cases officially provided by SPEC2006. Verification shows that *all* binaries instrumented by UROBOROS successfully pass all test cases, while two binaries (`gcc` and `mcf`) processed by DynInst fail the functionality testing.

Figure 4.7 presents the size increase for the function and the basic block level instrumentation. For both tests, DynInst increases binary sizes to more than twice of the original (131.2% increase for basic block counting and 119.1% for function counting), while UROBOROS only brings in less than 40% of size expansion for basic block counting and 2.0% for function counting. Note that for several cases, UROBOROS only introduces negligible size increase (`lbm` in both evaluations; `mcf` and `bzip` in function counting evaluation). It is worth mentioning that the instrumentation output of DynInst will load both the original and replica code into memory during run time. UROBOROS has a quite big advantage in terms of memory efficiency.

[5]Similar to other tools, UROBOROS cannot fully recognize all the functions in stripped binary code. We assume the function information is known to us in Section 4.4 and Section 4.5.

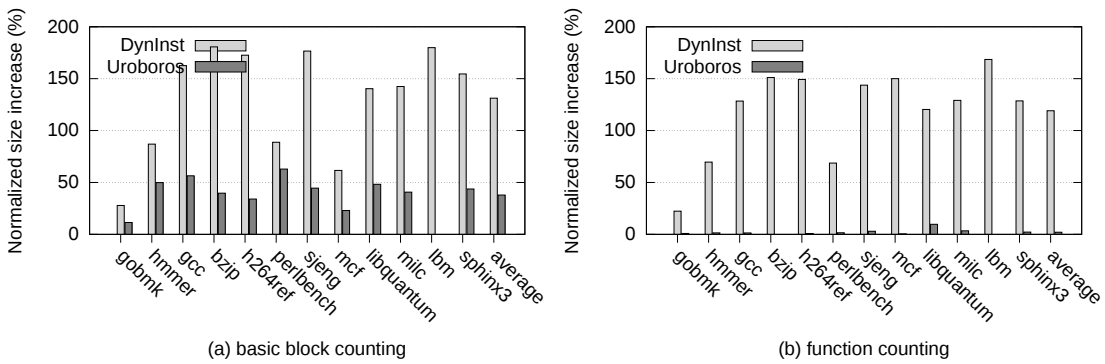


Figure 4.7: Size increase comparison.

Figure 4.8 presents the performance slowdown for the function and basic block level instrumentation. As DynInst enables the "replica-based" instrumentation, we expect that execution slowdown of DynInst could compete with the UROBOROS reassembly-based instrumentation. Indeed, for function level instrumentation, the average execution slowdown of DynInst (2.94%) is as good as UROBOROS (2.93%). For the basic block counting task, binaries instrumented by UROBOROS is slower than DynInst (93.38% compared to 76.56%). We attribute this to that DynInst adopts multiple customized optimization solutions. One such example is liveness analysis. The `inc` instruction used for counting basic blocks modifies CPU flags. In this experiment, we conservatively save/restore the flags in each instrumentation process. DynInst, on the other hand, performs liveness analysis to avoid some saving/restoring operations, offering an enhanced performance gain.

Still, we consider around 93% instrumentation slowdown is acceptable for instrumentation tasks with many targets such as basic block counting. On the other hand, as we reported, all the instrumented binaries of UROBOROS preserve the correct semantics, while two instrumented cases of DynInst (`gcc` and `mcf`) do not pass the functionality testing. The reason may because the DynInst instrumentation facilities contain certain bugs, or the employed rewriting strategy is likely to break the correctness when instrumenting complex binaries.

As aforementioned, previous instrumentation techniques significantly deforms binary structure, inevitably complicating subsequent analyses and transformations. In this experiment, we found that the commonly-used disassembler `objdump` reports a considerable number of decoding errors on the output from DynInst. In the

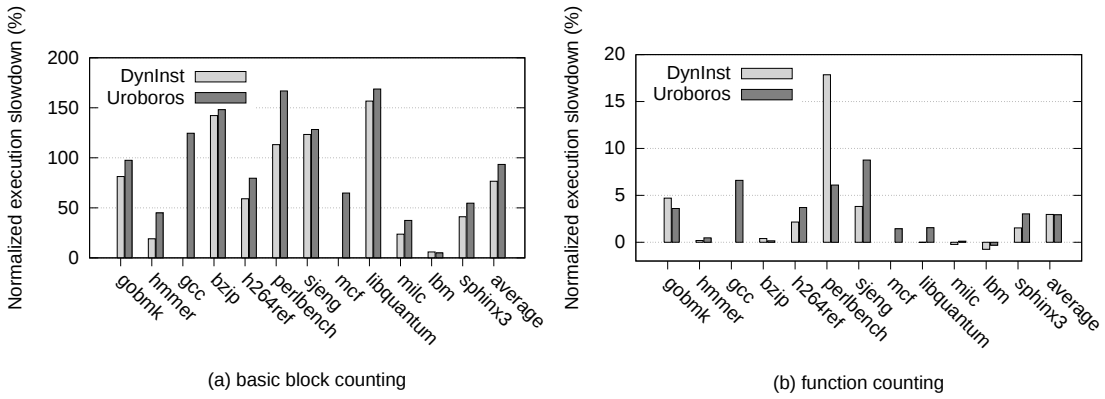


Figure 4.8: Performance slowdown comparison.

UROBOROS case, the instrumented binaries can be successfully disassembled by `objdump` with no error.

We also evaluate the processing time of UROBOROS with or without instrumentations. We design three tasks for evaluation, each of which instruments binaries at different levels of the program structure. We evaluate UROBOROS against the same set programs from 32-bit SPEC2006. The processing time is calculated from starting to disassemble the inputs until finishing assembling the instrumentation outputs.

We evaluate UROBOROS in terms of instruction, basic block, and function level instrumentations. Our instruction level instrumentation inserts "sandbox" instructions to all the indirect control transfers. An `and` instruction is used to validate the destinations of the upcoming control transfers. This instrumentation is quite useful in developing goal-oriented security applications, e.g., software fault isolation (SFI). The basic block and function level instrumentation inserts instructions at the beginning of each basic block or function to count the number of executed units.

Table 4.1 presents the evaluation data. The "baseline" column presents processing time without applying instrumentation while the other three columns show data with instrumentations. "Instr. A" corresponds to instruction level instrumentation; "Instr. B" and "Instr. C" represent basic block and function level instrumentations, respectively. On average, the instruction level instrumentation increases the processing time by 3.89%, basic block level instrumentation increases 14.50% while function level instrumentation increases 1.86%. The basic block level

Table 4.1: Instrumentation time evaluation.

Program	Baseline (s)	Instr. A (s)	Instr. B (s)	Instr. C (s)
bzip	0.72	0.77	0.87	0.77
hmmer	3.04	3.11	3.64	3.11
gcc	67.93	72.46	77.37	69.24
gobmk	31.75	32.04	35.89	32.39
h264ref	5.91	6.02	6.74	6.00
perlbench	12.92	13.04	16.43	13.08
sjeng	8.87	8.91	9.12	8.93
mcf	0.39	0.40	0.42	0.40
libquantum	0.62	0.62	0.69	0.63
milc	1.31	1.32	1.52	1.34
lbm	0.37	0.38	0.43	0.40
sphinx3	1.91	1.91	2.26	1.94
average	11.31	11.75	12.95	11.52

instrumentation has the most candidates to instrument, and therefore it imposes the highest instrumentation time cost.[6] Overall, the instrumentation processing time of UROBOROS is quite small.

4.5 Sample Applications

To demonstrate how UROBOROS can be used in practice, we developed two applications for UROBOROS. The first application, UroborosDiv, instruments the input binary for a large amount of iterations. An unique "iteration effect" is leveraged to boost the generation of large amounts of diversified binaries with increasing performance. The second application, UroborosTrace, records executed basic block information. It can be used for trace profiling. By demonstrating these applications, we show UROBOROS can perform some transformations that existing tools can hardly handle. For some of the transformations that can be done by other tools, we show that UROBOROS can result in better performance.

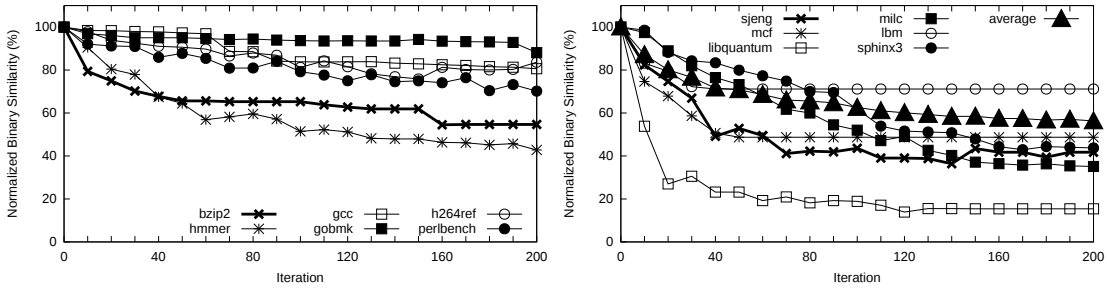


Figure 4.9: Binary similarity rates under different iterations.

4.5.1 Binary Code Diversification

Although it is originally proposed for optimization, function inlining has been employed as a software diversification strategy [50]. To demonstrate the strength of UROBOROS in binary instrumentation and transformation, we present a software diversification application UroborosDiv in this section. UroborosDiv is an instrumentation application that diversifies binaries by randomly selecting functions from a set of candidates and inlining them into their call sites. Different from the "one-off" design of most diversification frameworks, in which the original input is used to generate diversified copy one, two, three, etc., UroborosDiv takes the instrumentation output as the input and re-instrument it iteratively. As previously discussed, existing instrumentation tools, either patch-based or replica-based, can hardly process binaries in this iterative manner because massive binary structure changes impede secondary instrumentation. However, transformation through UROBOROS is *transparent*, i.e., UROBOROS produces *normal* binaries in which all newly inserted code is blended into the original code. Therefore, instrumented binaries can be directly re-processed by any existing binary tool, including UROBOROS itself.

We apply UroborosDiv to all the SPEC2006 C programs. By iterating the instrumentation procedure on a program binary, we can quickly produce a large number of unique copies. In the experiments, we iterate the pass for 200 times. After processing, we use test cases from 32-bit SPEC2006 to verify the functionality correctness on the diversified binaries, and all diversification outputs pass the functionality testing.

[6]Note that instruction level instrumentation only targets *indirect control transfer* instructions, while basic block level instrumentation targets *all* the basic blocks.

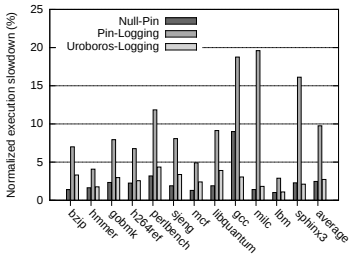


Figure 4.10: Performance overhead comparison for SPEC2006.

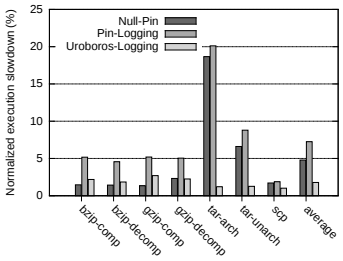


Figure 4.11: Performance overhead comparison for common utils.

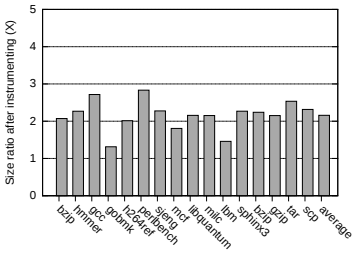


Figure 4.12: Size increase after instrumentation.

We evaluate the diversification effect by using the industry standard binary diffing tool BinDiff.[7] For each diversification output, we diff it with the original input and get the instruction level matching rate.[8] The evaluation data for all tested programs are shown in Figure 4.9. Binary similarity rate decreases for all the test cases. Note that as one function is inlined at each iteration, binaries with larger code sections, e.g., `gobmk`, `gcc`, get higher matching rates in general. It should be clarified that when this application is deployed in real-world scenarios, more iterations can be applied, also with more functions to be inlined in each iteration.

We have noticed a "stabling" trend in the case of `lbm`, `mcf` and `libquantum`. These binaries have relatively fewer functions, and it is likely that UroborosDiv has consumed all inline-able targets after certain iterations. Overall, as shown in the "average" data of Figure 4.9, average similarity rate decreases with more iterations. We interpret it as a promising result to show the diversification effect increases with more iterations. Again, the transparent instrumentation in Uroboros allows us to re-process the instrumentation output for many iterations. A large amount of instrumentation outputs can be generated, with the distinguished diversification effect enabled by iterations.

[7] `http://www.zynamics.com/bindiff.html`

[8] BinDiff provides the number of matched functions, basic blocks and instructions. Since function and basic block can be "partially" matched, e.g., 30% of the instructions in a function are matched with another function, counting the number of matched functions or basic blocks could be tricky. Therefore, we only adopt the instruction level matching rate.

4.5.2 Trace Profiling

In this section, we demonstrate the efficiency of UROBOROS by comparing it with Pin [41] (version 2.14, build 71313), on trace profiling, a widely-used instrumentation task. We iterate the basic blocks of a binary, and instrument each basic block with a sequence of instructions for logging. We record executed basic blocks by writing their memory addresses in the original binary to a global buffer. As for Pin, we write a simple Pintool (a tool built on top of Pin) to instrument basic blocks on the execution traces in the same way. We strictly follow the optimal Pintool writing strategies for better performance, such as Pin code inline, fast-call linkage, and `IPOINT_ANYWHERE` to schedule the call anywhere. Note that the profile data is not flushed to the disk, and the buffer will be rewritten when it is full. We consider the measurement without I/O overhead can give us a better estimation of the instrumentation cost. As discussed in Section 4.3.2.4, we avoid to save and restore CPU flags by misusing the `loop` opcode. Currently we allocate a 16M buffer to store the memory addresses of the executed basic blocks. We write a 4-byte memory address into the allocated buffer for each basic block.

Figure 4.10 shows the performance overhead of running 32-bit SPEC2006 C programs. We use the shipped test cases to measure performance. For each program, we instrument the binary with trace logging code and present the data in "UROBOROS-Logging" bar. The "Null-Pin" bar presents the Pin environment overhead, which runs the binary under Pin without any instrumentation. The "Pin-Logging" bar shows overhead when Pin is executed with our basic block instrumentation Pintool. On average, "UROBOROS-Logging" incurs 2.77X performance overhead comparing with the native execution, while "Pin-Logging" imposes as much as 9.76X overhead. In fact, the overhead of UROBOROS-Logging is close to that of "Null-Pin", which is 2.47X. Note that "Pin-Logging" can generate over 15X performance penalty for some test cases, e.g., `gcc`, `milc`, and `sphinx3`, while the overhead of "UROBOROS-Logging" is relatively stable at around 2–4X. In fact, the average performance overhead of "UROBOROS-Logging" reaches the theoretically lowest value. Particularly, as on average a basic block has 5–10 instructions, and for each basic block, UroborosTrace inserts 7 new instructions, the total amount of executed instructions could double. Furthermore, as the `loop` instruction has a relatively high CPU ticks (7 for our experiment platform), we estimate the

theoretical lowest execution slowdown is around 2.3–2.7X, which matches very well with "UROBOROS-Logging" (2.77X).

We also evaluate UroborosTrace on four common Linux utilities which represent three kinds of workload. The program `tar` is I/O bound, `bzip2` and `gzip` are CPU intensive program, and `scp` is a network application. We use `tar` to archive and extract the GNU Core utilities 8.13 package (about 50MB). The same input is used by two compressors for compressing and decompressing and `scp` sends the archived package through a 1 Gbps Ethernet link. Figure 4.11 shows the evaluation data. The CPU bound programs show similar evaluation results with Figure 4.10. UroborosTrace imposes relatively very low overhead on two tests of the `tar` program (on average 1.23X), while on average over 14.5X overhead is incurred by Pintool. Note that different from Figure 4.10, the average data of "UROBOROS-Logging" is even better than "Null-Pin". Overall, UroborosTrace exhibits a decent runtime performance.

Figure 4.12 presents the size increase of instrumented binaries. The average size increase is 2.16X, which matches well with our expectation. In summary, UROBOROS is quite *efficient* compared with real-world dynamic instrumentation tools.

Chapter 5
Function Recognition in Binary Executables

Function information is critical for many binary code retrofitting and analysis tasks. Nevertheless, function symbols are usually stripped from binary executables to defeat adversary analysis before release. By far, identifying functions in stripped binaries remains a challenge. Recent research work proposes to recognize functions in binary code through data mining techniques. The recognition model, including typical function entry point patterns, is automatically constructed through learning. However, we observed that as previous work only leverages *syntax-level* features to train the model, binary obfuscation techniques can undermine the pre-learned models in real-world usage scenarios.

In this chapter we propose FID, a *semantics-based* method to recognize functions in stripped binaries. We leverage symbolic execution to generate semantic information as the training set and learn a function recognition model through well-performed data mining techniques. We start by introducing the research context and giving a motivating example in Section 5.1. We then present the design of FID in Section 5.2 and iterate each component of FID in the following subsections. Section 5.3 reports the evaluation results regarding benign binaries as well as obfuscated code.[1]

[1]The work of this chapter is published in the 33rd IEEE International Conference on Software Maintenance and Evolution [78].

5.1 Introduction

Function recognition in binary programs is critical in reverse engineering [21, 79, 80] and many binary instrumentation and analysis tasks [12, 81, 82]. For example, control-flow integrity validates control flow transfers with rules constructed before execution [6,13,83], and function addresses are used to define these rules. In addition, many binary similarity analysis tools launch the similarity test in the granularity of functions [84–86]. Thus, incorrectly identified functions can drastically impede the similarity test. Recent research work [40] studies the recovery of relocation information from binary code, in which function addresses are the prerequisite to identify code pointers.

Despite the fundamental role of functions in binary instrumentation and analysis applications, function information is usually absent in real world program binaries. The main reason is that to reduce application size and defeat reverse engineering from adversaries, program symbols (including function information) are usually removed from program binaries before distribution. By far, identifying functions in stripped binaries remains a challenge. Some research work has been proposed to discuss the recognition of functions in stripped binaries [63, 64]. Also, most of the widely-used binary reverse engineering and analysis tools have implemented their own methods to identify functions [25, 26, 87]. Note that most of these existing tools rely on handwritten patterns to recognize function *prologue instructions*. However, it is reported that these manually written patterns can become less effective when the input binary is highly optimized. Indeed, it has been pointed out that the industry strength reverse engineering tool IDA-Pro (version 6.5) which features function detection fails to recognize functions in a simple C program compiled by Intel `icc` compiler with the O3 optimization level [68].

Distinguished from these manually written patterns, Rosenblum et al. [65] first consider the function recognition as a data mining problem; patterns are automatically learned from the training data, which will be installed for usage. In addition, recent work proposes advanced data mining techniques to recognize functions with improved performance [68, 69]. In general, these data mining methods automatically learn key features from a large set of binary code to train a detection model, and given a sequence of machine code bytes (or assembly instructions), the "learned" model is able to answer whether the given code bytes

start new functions. The data mining approaches have been evaluated to work better than handwritten pattern matching methods. However, we observe one of the limitations of these methods is that they construct the "key features" purely through machine code bytes (or assembly instructions). That means, the leveraged features only capture the *syntax-level information* in the binary code. Thus, it is reasonable to suspect that program syntax changes can potentially defeat the learned models due to different complication settings or even program obfuscations.

Program obfuscation and diversification transform programs into complex representations which are difficult to understand. Typical obfuscation techniques insert garbage code into random positions of the program, change the control flow structures, and harden control flow predicates into opaque formats. To defeat reverse engineering and analysis from adversaries, besides deleting the debug and relocation information, we assume *software can be obfuscated* before release. To provide better analysis facilities of real world program binaries, we study the function recognition problem against binary obfuscated by commonly-used techniques, which, to our best knowledge, has not been evaluated by previous work systematically.

Symbolic execution captures the semantic information of programs. The key idea of symbolic execution is to use symbolic variables to represent the input and (statically) interpret the code. After symbolic execution, for each initial input, a symbolic formula is generated to represent its output semantics. To better tackle the function recognition problem, we propose to identify functions through the combination of symbolic execution and data mining. To this end, we first employ an open-source reverse engineering tool Uroboros [40, 75] to disassemble the input binary and recover basic blocks. We then apply symbolic execution on *each individual* basic block to generate corresponding semantics. In particular, we record the assignment formula ([85]) of each register which captures the behavior within one block as well as memory accesses during the interpretation. We select key semantics from the outputs of symbolic execution, and translate them into numeric feature vectors. We utilize well-performing data mining techniques to learn from the acquired key features and train a recognition model. For any given basic block, the learned model can answer whether it represents a *function entry point basic block* or not, thus identifying a new function. We implement the proposed technique in a tool named FID, and we evaluate FID against a broad set of diverse program binaries produced by three compilers and four optimization levels. The evaluation

results show that FID is comparable or even outperforms the state-of-the-art function recognition tools towards the broad sets of test cases we use. We also employ a widely-used program obfuscation tool, Obfuscator-LLVM [88] to measure the obfuscation resilience of FID (Obfuscator-LLVM is referred as `ollvm` later). Binaries transformed by seven obfuscation strategies are produced and evaluated in this paper (including three widely-used binary obfuscation methods and their four compositions). Our evaluation shows that while previous tools suffer from the drastically changed syntax in obfuscated binary code, the performance of FID is quite promising. In sum, this paper makes the following contributions:

- We identify the limitations of previous data mining based techniques in function recognition, i.e., model learned from syntax-level features can be defeated easily by program syntax changes. We propose a novel technique to extract the semantics and learn a more robust model. We implement our proposed approach as a practical tool, FID.
- We evaluate a broad set of normal and obfuscated program binaries. Evaluation shows that our approach can successfully capture the semantic information across various compilers and optimization levels. Our evaluation also reports that FID can outperform previous available tools against multiple widely-used obfuscation transformations and their compositions.

5.1.1 Motivating Example

We observed that previous function recognition methods can become malfunctional in front of program syntax changes. We present an example in Figure 5.1, in which a data mining based function recognition tool ByteWeight [68] misidentifies a function entry point.

To set up this test, we first compile all the 32-bit GNU Coreutils binaries (version 8.23) using `LLVM` 3.6 and optimization O0. We train ByteWeight (bap-byteweight in BAP v0.99 [89]) to learn a recognition model from all these binaries. This version of ByteWeight captures informative machine code bytes to train the model. ByteWeight also has another implementation which takes assembly instructions to train the model [90]. In the rest of this paper we refer to the byte-level ByteWeight as BW-Byte while the other one as BW-Instr.

```
1  <emit_try_help>:
2   push   %ebp
3   mov    %esp,%ebp
4   sub    $0x18,%esp
5   lea    0x804db36,%eax
6   mov    0x8050144,%ecx
7   mov    %eax,(%esp)
8   mov    %ecx,-0x4(%ebp)
9   call   gettext
```

(a) Original code

```
eax = 0x804db36
ebx = reg2
ecx = mem1
edx = reg4
esi = reg5
edi = reg6
ebp = reg8
esp = reg7 - 32
```

(b) Assignment formulas corresponding to the original code

```
1  <emit_try_help>:
2   push   %ebp
3   nop
4   mov    %esp,%ebp
5   sub    $0x18,%esp
6   lea    0x804db36,%eax
7   mov    0x8050144,%ecx
8   mov    %eax,(%esp)
9   mov    %ecx,-0x4(%ebp)
10  call   gettext
```

(c) Obfuscated code

```
eax = 0x804db36
ebx = reg2
ecx = mem1
edx = reg4
esi = reg5
edi = reg6
ebp = reg8
esp = reg7 - 32
```

(d) Assignment formulas corresponding to the obfuscated code

Figure 5.1: A motivating example.

Garbage code insertion obfuscates programs by inserting meaningless instruction sequences into the code. To present a straightforward and informative example, We insert garbage code to obfuscate the syntax of one Coreutils program binary (`basename`). To this end, we disassemble the program binary of `basename`, insert one `nop` instruction at the beginning of the function `emit_try_help` (line 3 in Figure 5.1c), and reassemble the instrumented output into an executable. Figure 5.1a presents the prologue instruction sequence of `emit_try_help` before obfuscation,

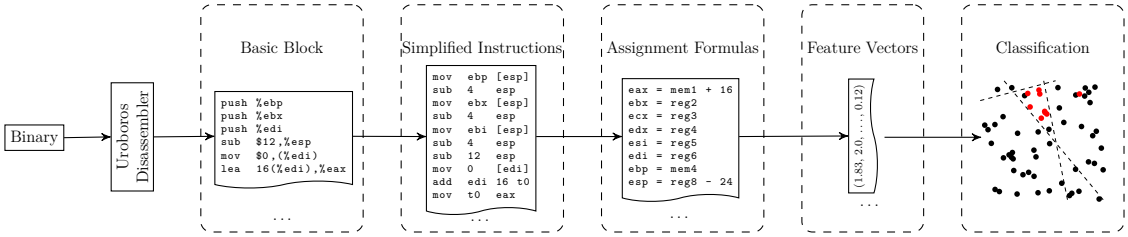


Figure 5.2: The workflow of FID.

and Figure 5.1c shows the obfuscated code. We then employ BW-Byte to identify functions from both the original and the obfuscated binaries; we report that while BW-Byte can correctly recognize this `emit_try_help` function from the original binary, the same function cannot be recognized from the obfuscated code.

BW-Byte constructs weighted prefix trees to represent typical function entry point patterns, each tree node maintaining one machine byte. We consider the main reason for the misidentification is because the inserted `nop` defeats the matching towards the pre-learned tree structures. To illustrate the hidden similarity between the original and obfuscated code, we present the *assignment formula* of eight registers through symbolic execution ([85]). Assignment formulas capture the code semantics in terms of input and output relations. We initialize each register with a symbolic variable as the input (e.g., `reg1`), and during interpretation, every memory access towards uninitialized region (e.g., line 6 in Figure 5.1a) creates a new symbolic variable as well (e.g., `mem1`). The interpretation outputs are shown in Figure 5.1b and Figure 5.1d; the assignment formulas of the original and obfuscated instruction sequences—as can be expected—are equivalent. Note that while most formulas have one symbolic variable, formula of stack register `esp` contains a subtraction operation of 32. Typically for programs on the x86 architecture, stack needs to grow to store local variables, which decrements the stack register. In sum, the recognition model trained from program syntax could become *unreliable* in front of even simple syntax changes, while the semantics can usually be preserved regarding such changes.

5.2 Design

We now outline our approach for function recognition in program binaries. To this end, we train a classifier through semantics of function entry point basic blocks.

Later, for a given basic block, our classifier is able to answer whether it represents a function entry point or not, thus recognizing a new function. The extracted semantics is mainly represented as assignment formulas, which describe basic block's behavior regarding the input and output relation of registers.

FID is built on top of UROBOROS, an open-source binary disassembly and instrumentation platform [40]. The input program binaries are disassembled and maintained as its internal data, and UROBOROS provides utilities to perform inspection and manipulation. While some program instrumentation facilities are provided already, the analysis component of UROBOROS is quite insufficient. In this paper, we extend UROBOROS with multiple analysis functionalities.

Scope and Assumptions. FID is mainly designed to recognize functions inside x86 ELF binaries without debug or relocation information. We evaluate it in test cases compiled by different compilers, optimization levels and commonly-used obfuscations. Careful readers may notice that FID extracts semantics of each basic block. Thus, correct disassembling and basic block recovery are the prerequisites of FID. In this paper, we assume the disassembling and basic block recovery are mostly reliable.

As previously mentioned, one motivation of our research is that syntax-based pattern is untenable or even misleading due to syntax changes. Therefore, in this research, besides `call` instructions identified inside the code section of the program binary, FID does not take program syntax into consideration. In addition to the trained recognition model, the destinations (i.e., the callee) of `call` instructions are used to reveal more functions.

5.2.1 Workflow

Figure 5.2 shows the overall workflow of FID. FID takes a stripped binary as input and employs UROBOROS to recover program control flow structures (including each basic block). FID visits each recovered basic block and launches the implemented analysis components.

As UROBOROS is mainly designed to support binary instrumentation, assembly instructions maintained by it are not parsed into *analysis-friendly* formats. Therefore similar to other binary analysis tools, FID first simplifies the complex representations maintained by UROBOROS and lift them into easy-to-analysis for-

mats. This process exposes instructions with implicit memory operations into corresponding explicit expressions and simplifies composite instruction operands (e.g., indirect addressing).

In the next phase, we launch a symbolic execution engine to produce the semantics of each basic block. Note that as we are only analyzing each individual block, it is not necessary to track the intra and inter-procedure execution information. We capture the *assignment formulas* for eight general-purpose registers and also record the memory access behaviors in this step (Section 5.2.2). Given all the acquired semantics, we then select *informative* features and trim off redundancy before learning (Section 5.2.3). We emphasize this step is necessary as the deliberately-selected key features can boost the learning process and improve the performance.

We now have the representations of the semantics each basic block has; the assignment formulas and memory access behaviors describe operations a basic block will perform. However, assignment formulas are purely *syntactic*; learning directly from assignment formulas are very challenging. Therefore, we then seek to translate semantics into *numeric feature vectors*. We extract multiple numeric features from both *lexical* and *syntactic* aspects of an assignment formula (Section 5.2.4).

With all the numeric vectors collected, we then discuss how we launch the learning process and train the recognition model (Section 5.2.5). In addition, our study shows that binary compiled by different compilers may have different feature distribution, and to present a practical tool, we undertake a *pre-classification* step, determining which compiler the input binary is compiled from (Section 5.2.6). After that, for a given binary, we use the model learned from binaries compiled only by the identified compiler to predicate.

Similar with previous work [68, 70], we improve the accuracy through control flow analysis, i.e., identifying function call instructions in the disassembled code. Motivated by the low recognition precision of previous tools (Section 5.3.3), one of our central design choice is to present *conservative* feature selection which guarantees high precision rate, and improve the recall rate with *call instruction collection* (Section 5.2.7).[2]

[2]In this paper, *precision* represents the percentage of function entry points identified that are correct; *recall* is the percentage of real function entry points identified as such. We also calculate *F1 score* in evaluation sections, which is the harmonic mean of precision and recall; naturally, the higher F1 score is, the better a learned model is considered in general.

5.2.2 Basic Block-Level Symbolic Execution

After acquiring the simplified code, the first step is to leverage symbolic execution to interpret instruction sequences corresponding to each basic block. For each basic block, we collect the assignment formula of every 32-bit general-purpose register. We also record the memory access behavior a basic block commits during the execution. As mentioned previously, symbolic execution engine implemented in FID initializes the interpretation at the beginning of every basic block; this design choice can largely improve the practicability, as typical challenges in analyzing real-world large size binaries, such as inter-procedural analysis, are not considered. Our implementation follows the common design of a symbolic execution engine. We leverage bit vectors defined by the Z3 SMT solver [91] to construct the input value of each register. The symbolic variables initialized through Z3 allow bit-level arbitrary computation, and after aggregating assignment formulas representing the output semantics, we pass formulas to the Z3 solver to simplify the expressions before further analysis.

FID also maintains a lookup table to represent memory contents and how each position is accessed through memory addressing formulas (e.g., `4*reg1 + 4`, where `reg1` is the input variable of register `eax`). Memory read operations on stack is likely to indicate access on function parameters, which is considered as a key feature of function starting blocks (see Section 5.2.3 for details). Therefore after execution, FID iterates each recorded memory read operation, and if any of the memory read formula contains the input symbolic variable of the stack register `esp`, this memory access formula will be recorded. Note that it is likely to use other registers instead of `esp` to access stack, as we statically interpret the instructions and keep track of the memory, dataflow from `esp` to other registers can be captured as well. After the interpretation, assignment formulas and memory read formulas of each basic block are dumped out for further analysis.

5.2.3 Select Informative Semantics

Before we "learn" a model from all the acquired data in Section 5.2.2, we first select the key features that are mostly informative in this research. Indeed, our preliminary test shows that by deliberately selecting a subset from all the outputs in Section 5.2.2, we could notably improve the performance of FID.

```
mov   $0x805248e,0x4(%esp)
mov   -0x10(%ebp),%eax
mov   %eax,(%esp)
call  c_strcasecmp
```

(a) Caller's basic block

```
eax = mem1
ebx = reg2
ecx = reg3
edx = reg4
esi = reg5
edi = reg6
ebp = reg7
esp = reg8
```

(b) Assignment formulas of the caller's basic block

```
<c_strcasecmp>:
 push   %ebp
 mov    %esp,%ebp
 push   %esi
 push   %ebx
 sub    $0x20,%esp
 mov    0x8(%ebp),%esi
 mov    0xc(%ebp),%ebx
 cmp    %ebx,%esi
 jne    0x804cf85
```

(c) Callee's basic block

```
eax      = reg1
ebx      = mem1
ecx      = reg3
edx      = reg4
esi      = mem2
edi      = reg6
ebp      = reg8 - 4
esp      = reg8 - 44
[reg8+4] = mem3
[reg8+8] = mem4
```

(d) Assignment formulas and memory reads of the callee's basic block

Figure 5.3: Memory access behaviors in an inter-procedure control transfer.

Stack Registers. We observe that most function entry point basic blocks will create new stack frame and reserve spaces to allocate local variables. That means, stack register `esp` and `ebp` are likely to be adjusted in typical function entry point basic blocks. Figure 5.4 presents an example, demonstrating how stack register `esp` is used in a typical function beginning basic block. As shown in Figure 5.4a, register `esp` is utilized to reserve 92 bytes on the stack to store local variables. Note that besides the explicit subtraction operation on `esp` (line 5 in Figure 5.4a), `push` opcode (line 1–4) also implicitly decrements the input value of `esp` by 16 bytes

```
1   push   %ebp
2   push   %ebx                    eax = mem1
3   push   %edi                    ebx = reg2
4   push   %esi                    ecx = reg3
5   sub    $0x4C,%esp              edx = mem2
6   mov    0x68(%esp),%eax         esi = mem3
7   mov    0x64(%esp),%edx         edi = reg6
8   mov    0x60(%esp),%esi         ebp = reg7
9   test   %edx,%edx               esp = reg8 - 92
```

(a) A function entry point basic block (b) The corresponding assignment formulas

Figure 5.4: Stack register adjustments in a function entry point.

(this implicit effect has been translated into explicit statements before symbolic execution). Figure 5.4b presents the assignment formulas corresponding to instructions in Figure 5.4a. Assignment formulas of register `esp` contain a subtraction operation of 92. In general, stack registers are majorly *informative*, and we pick their assignment formulas (formulas of register `esp` and `ebp`) to learn.

Memory Read. Although stack registers are informative, actually in a typical function call context, both caller and callee can manipulate the stack registers. To future distinguish caller and callee, we elaborate on how we select key features from memory access behaviors.

In this research, we capture the stack memory *read* operations through register `esp`, which is likely to indicate typical parameter read operations at the beginning of functions. Figure 5.3 presents typical memory access instructions in a function call context (this example is from a GNU Coreutils program `printf`). As shown in Figure 5.3d, memory positions pointed by `reg8+4` and `reg8+8` which represent the memory positions of the first two function parameters, are all visited by the callee (`reg8` is the input variable of stack register `esp`). Whereas on the caller side no memory read can be found.

In general, we assume stack memory read operations are informative in identifying function entry points, if the memory access formula follows certain addressing patterns. Recall in Section 5.2.2 the symbolic execution engine has iterated all

the memory read addressing formulas and dumped out formulas containing the input variable of `esp`. We then check the presence of memory addressing formulas following such pattern `reg8 + 4*n`, where n can equal to 1, 2, and 3 (suppose `reg8` is the input variable of `esp`, and we assume each stack memory fetching is 4-byte aligned). Such addressing formulas indicate memory read towards the first three potential parameters of a function. Functions with three or more parameters will have the same feature vectors (i.e., `(Present, Present, Present)`) in this step.

5.2.4 Translate Assignment Formulas into Numeric Vectors

Several previous work seeks to recognize equivalent assignment formulas through a theorem prover [92, 93]. Given two formulas, a theorem prover is able to prove the equivalence between them, thus identifying program units (e.g., two basic blocks) that are semantically equivalent. However, despite its disinformative results (a prover can only tell "match" or "unmatch"), we emphasize this equivalence-seeking approach is not suitable in our usage scenario, as we are more interested in the gradual similarity.

On the other hand, we observe that most data mining methods take *numeric vectors* to train the model. Enlightened by recent research [94], we seek to translate the acquired semantics into numeric vectors to support a forthright learning process. We also notice that some machine learning algorithms are able to use more complex representations (e.g., string and tree kernels [95]). We leave it as one future work to explore challenges in adopting such advanced models in mining symbolic formulas.

In this step, we choose to extract and combine *lexical* and *syntactic* features from assignment formulas; each feature as a *numeric value*. Most lexical features are captured directly from formulas' textual representations, while syntactic features are acquired from the parsed abstract-syntax trees (ASTs). Besides, we also capture three boolean (0/1) *stack* features based on the stack memory access behaviors. Note that each assignment formula produces 8 features, and we capture formulas of `esp` and `ebp` (Section 5.2.3). Thus, for each basic block, we construct a numeric vector with 19 elements (8*2+3).

Lexical Features. Table 5.1 shows the lexical features we extract from the textual representations. We obtain the number of operators and constants by directly analyzing the text. As for the token related feature, we employ Python library

Table 5.1: Lexical features.

Feature	Definition
numOperator/length	the number of occurrences of operators divided by the formula length of characters
numToken/length	the number of tokens divided by the formula length of characters
numConstant/length	the number of constants divided by the formula length of characters
decOperator/length	the number of subtraction operators divided by the formula length of characters
decNum/length	the number of "small operands" in subtraction operations divided by the formula length of characters

Table 5.2: Syntactic features.

Feature	Definition
maxNestingDepth	the maximum levels of nested parentheses
maxDepthASTNode	the maximum depth of an AST
aveTreeDistance	the average tree edit distance between the target AST and 50 random picked ASTs

`tokenize` to calculate the total number of tokens. We are particularly interested in the subtraction operations of stack registers in typical function entry point basic blocks, and to this end, we identify two features regarding the presence of subtraction operations and their (potentially) small operands.[3]

Syntactic Features. In this step, we extract features that can be ignored in the lexical analysis. In particular, as each assignment formula can be parsed into a *syntax tree*, we extract syntactic features on top of the parsed tree. Table 5.2 presents features we utilized; We calculate the maximum levels of nested parentheses and the maximum depth of an AST as two features. Considering function prologue block delivers unique assignment formulas, it shall be accurate to assume ASTs of prologue blocks and other blocks would yield different similarity distributions when comparing to randomly selected ASTs. Hence, we calculate similarity for each AST with other randomly select ASTs in our dataset. *Tree edit distance* is employed to measure the similarity, and we use Python library `zss`, which implements the Zhang-Shasha algorithm [96] to calculate the distance. Without losing generality, we randomly select 50 ASTs and calculate their tree edit distances towards one target AST.

Stack Features. As in the previous step we have checked the presences of three stack memory accesses that indicate function parameter read (Section 5.2.3), here we create three boolean (0/1) features for them. Zero indicates the absence and one is for the opposite.

[3] "Small operands" refer to operands of subtraction operations that are less than a threshold. In our prototype implementation this threshold is 65536.

Table 5.3: Classifiers used by the majority voting mechanism.

Models	Settings
LinearSVC	penalty parameter C=16.0
AdaBoost	number of weak learners=100
GradientBoosting	number of boosting stage=100; learning rate=1.0; random state=0

Normalization

Although it may not be obvious, the *lexical* and *syntactic* feature extractions have implicitly "normalized" the assignment formulas (normalization here means generalizing a formula so that it can match formulas with similar structures). The reason is that we extract structural and tokenism information from the formulas (e.g., the total number of constant numbers), and ignore concrete values in the formula. The "normalization" can usually help us identify more functions. For example, if one function entry point basic block has an assignment formula `esp = reg8 - 4 + mem4`, then it is reasonable to assume a basic block with formula `esp = reg3 - 8 + mem5` belonging to a function entry point as well since two formulas produce the same lexical and syntactic features.

Even though we distinguish ourselves from previous work as we extract features from the semantics while they focus on syntaxes, normalization is considered as a general optimization for both.

5.2.5 Classification

After producing all the numeric feature vectors, we then employ data mining methods to train a recognition model. Our preliminary test shows that multiple data mining techniques have good performance. To present a well-performing and robust classifier, we decide to employ the *majority voting* mechanism on top of multiple learning algorithms. A typical majority voting classifier combines multiple learning methods and use a majority voter to predict. A majority voting approach can rule out weakness of each individual method, which should be more adaptable in our scenario.

As shown in Table 5.3, our majority voting classifier contains three classic learning methods. Besides settings shown in Table 5.3, we use the default value for all the other parameters of the three learning methods. We use these methods from Python data mining library `scikit-learn`.

5.2.6 Distinguishing Different Compilers

Our preliminary test on three compilers (`gcc`, `icc` and `LLVM`) shows that while binaries compiled by `gcc` and `icc` share quite similar feature distribution, `LLVM` has a slightly different distribution (Section 5.3.1). Given this observation, we construct a pre-classification step before predication in the real-world usage scenarios, determining whether a given program binary is compiled by `LLVM` compiler or `gcc/icc` (we put binaries compiled by these two compilers into one group as they have similar feature distributions according to our observation). After that, we recognize functions from the input with a model trained from binaries compiled only by the corresponding compiler (i.e., `LLVM` or `gcc/icc`).

The question we seek to answer in this step is comparable to a classic pattern recognition problem, i.e., given an image with thousands of pixels (in our case, it is compared to a program binary with thousands of basic blocks), which category does this image belong to. Enlightened by research work in that field, we classify program binaries according to the feature *distribution* of informative basic blocks. We first select representative basic blocks from thousands of candidates each binary contains, and then calculate the distribution of these selected blocks regarding some revealing features. We use the acquired distributions (in the format of numeric vectors) to train a classification model. After the training step, for a given binary code (i.e., the *input*), our classification model is able to answer which compiler it is compiled from (the *output*). We elaborate on our method in terms of a three-step approach as below:

The first step is to select "critical" basic blocks from all the blocks a binary contains. Naturally, basic blocks corresponding to function entry points should be considered as informative candidates in our research. However, our study has shown that without knowing which compiler it belongs to, identifying functions from obfuscated binaries through pre-learned models could become problematic (low recall rate in our case). Thus, we conservatively select functions identified by the callees of `call` instructions (as shown in Table 5.7, we can discover about 50% functions through this method).

The semantics of each basic block is translated into a feature vector with 19 elements (Section 5.2.4); as most of them are *continuous* variables, it is—if possible at all—quite challenging to calculate a feature distribution. Instead, as shown

Table 5.4: Boolean features to distinguish different compilers.

Feature	Definition
decESP	Does `esp` contain subtraction operator?
decEBP	Does `ebp` contain subtraction operator?
memREAD	Can we identify stack memory read operations on memory position `reg8` + 4*n (n can equal to 1, 2 and 3)?

in Table 5.4, the second step constructs five boolean features (note that `memREAD` stands for three features, and `reg8` is the input variable of register `esp`) from the numeric features we already acquire (Section 5.2.4). As these features are all in boolean distribution, the overall distribution space (2^5) is small and practical. We then calculate the distribution of function starting basic blocks with respect to these 2^5 (i.e., 32) variants and train a model using a majority voting classifier with the same settings (Section 5.2.5). Evaluations in this step are detailed in Section 5.3.2.

5.2.7 Call Instruction Collection

Our preliminary study shows that when analyzing complex binary code, it is *not always possible* to achieve low false positive and negative rate at the same time. Besides, we have observed that some of previous data mining based tools can have relatively high recall rate with quite low precision rate (Section 5.3.3). This is not satisfying in developing security applications (e.g., control-flow integrity [6, 12, 13]), as low precision rate indicates many instructions are incorrectly considered as function entry points, which potentially leaves more opportunities for attackers to hijack the control flow.

To present a practical tool, one of our central design choice is to preserve low false positive rate through deliberately-selected *conservative* features, and reduce the false negative rate with additional control flow analysis. To catch functions that are missed by the trained model, we extend the function entry point list through *call instruction collection*. That is, for a given call instruction, if we can identify its operand (say, the callee) in the code section, the callee is considered a new function beginning. Functions identified by this approach will be added to the final result of FID, thus reducing the false negative rate. We name this technique FID-CIC later in this paper.

5.2.8 Function Boundary Identification

Naturally, after recognizing function entry points, the next step is to recover the function boundary, i.e., identifying both the entry point and (multiple) exit points of functions. Most of the existing work recovers the function boundary information through control flow analysis [68, 70]; starting from the identified function entry point, they traverse the intra-procedural control flow to rebuild the control flow graph, thus recovering the function boundaries. Actually given the identified function entries, as the intra-procedural CFG recovery techniques are mostly well-developed, function boundary identification is a matter of *engineering effort*. So our major effort in this paper is to present novel techniques in recovering the function entry points.

ByteWeight adopts one baseline method to split the whole code section into multiple regions according to the identified function beginnings; each region stands for one function. Indeed besides typical challenges such as overlapping functions in highly-optimized binary code, this "naive" method has been proved as quite reliable [68]. FID provides this method to recover the function boundary. Indeed, we can utilize the *value-set analysis* to recover indirect control destinations and precisely reconstruct the CFG [97]. Thus, the function boundaries can be distinguished even for overlapped functions. We leave it as one future work to extend FID with the precise recovery of function boundaries.

5.3 Evaluation

We undertake a three-step evaluation in this research. The first step evaluates FID in function entry point recognition of normal binaries; three compilers and four optimization levels are employed to generate test cases in this step. We then test FID in distinguishing which compiler an input binary is compiled from. The third evaluation is on obfuscated code. We leverage the Obfuscator-LLVM [88] (referred to as `ollvm`) to obfuscate all the test cases with three widely-used obfuscation methods and their compositions (in total 7 strategies). All the optimization levels are used to generate obfuscated code in this evaluation. We evaluate FID in terms of three standard criteria, i.e., *precision*, *recall*, and *F1 score*. In general, our experiments aim to address the following questions:

- Is FID resilient to compiler and optimization changes (Section 5.3.1)?
- Is FID capable of answering which compiler an input binary is compiled from (Section 5.3.2)?
- Is FID resilient to widely-used code obfuscation techniques (Section 5.3.3)?

Before we present the evaluation of our approach, we first introduce the data set we use and how we acquire the ground truth for comparison.

Data Set. Our evaluation are designed to compare with the cutting-edge research and industrial binary analysis tools who features the function recognition functionality. We choose to employ a widely-used program set, i.e., GNU Coreutils as the test set in our research. GNU Coreutils consists of 106 binaries which provides diverse tasks on Linux operating systems such as textual processing, system management, and arithmetic calculation. We compile the test cases with three compilers (`gcc`, `icc` and `LLVM`) and four optimization levels to produce "normal" program binaries for evaluation.

To evaluate the resilience to program obfuscation, we employ `ollvm` in our experiments. `ollvm` is a set of obfuscation passes implemented inside `LLVM` compiler suite, which provides three widely-used obfuscation methods, i.e., instruction replace [48], opaque predicate insert [62], and control-flow flatten [50] to obfuscate the inputs. All of these methods are widely-used in typical program obfuscation tasks. In this paper, we leverage all the implemented obfuscation methods, together with their *compositions* (i.e., combining multiple methods to obfuscate) to produce binaries with complex structures. We present the obfuscation strategies we use in Table 5.5. Note that each column name corresponds to the abbreviated name we use in evaluation. In summary, our data set consists of three variables:

Compiler. We use GNU `gcc` 4.7.2, Intel `icc` 14.0.1 and `LLVM` 3.6 to produce test binaries.

Optimization Level. For both "normal" and obfuscated binary evaluation, we test all the optimization levels, i.e., O0, O1, O2 and O3.

Obfuscation Methods. We test program binaries obfuscated by 7 different strategies; each strategy is evaluated regarding 4 optimization levels as well.

In total, 4,240 (1,272 normal binaries and 2,968 obfuscated binaries) unique test cases are evaluated in our work.

Table 5.5: Obfuscation strategies used in the evaluation.

Obfuscation Methods	ins	opq	flt	mix1	mix2	mix3	mix4
Instruction Replace	✓			✓	✓		✓
Opaque Predicate Insert		✓		✓		✓	✓
CFG Flatten			✓		✓	✓	✓

Table 5.6: Ten-fold validation on different compilers with different optimization levels.

Opt. Level	LLVM			gcc			icc		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
O0	0.828	0.980	0.898	0.961	0.978	0.969	0.961	0.979	0.970
O1	0.868	0.933	0.899	0.958	0.951	0.954	0.958	0.952	0.955
O2	0.792	0.961	0.868	0.957	0.946	0.951	0.957	0.945	0.951
O3	0.826	0.961	0.888	0.961	0.955	0.958	0.961	0.857	0.906
average	0.829	0.959	0.889	0.959	0.958	0.958	0.959	0.933	0.946

Ground Truth and Tool Usage. All the test cases are compiled with the symbolic and debug information, and it is easy to get the ground truth (i.e., function beginning addresses) by disassembling the binary. Indeed we acquire the ground truth by disassembling the code section of each test case with GNU tool `objdump`, and extract all the functions with their starting address information by using `grep`.

The symbolic and debug information will then be removed from test binaries using GNU tool `strip` before analyzed by FID. Note that while FID and ByteWeight can directly output the identified function starting address, IDA-Pro recovers functions as its internal data structure for analysis and transformation. For IDA-Pro, we write scripts to dump out the function information. As previously mentioned, ByteWeight provides two syntax-based function recognition methods which generate machine byte or assembly instruction-based models. Both of them are evaluated in our research. we summarize the tools we evaluated below:

BW-Byte: The machine byte-level ByteWeight has been integrated into BAP [26]. We use BAP version 0.99 for evaluation (the newest version by the time of writing) [89].

BW-Instr: The instruction-level ByteWeight is provided through a virtual machine image [90]. The image is downloaded and configured to use.

IDA-Pro: We use IDA-Pro version 6.6 with all the function identification options enabled.[4]

[4]Although the newest version of IDA-Pro is 6.9 by the time of writing, there is no improvement for function identification in x86 ELF binaries according to its release notes [98–100].

5.3.1 Normal Code

We first evaluate FID in all the normal programs. As previously mentioned, we utilize three compilers and four optimizations to compile programs in GNU Coreutils, resulting into 1,272 test cases. To demonstrate the proposed semantics-based model in FID, *call instruction collection* (Section 5.2.7) is *not* used in this step.

We used ten-fold validation in this step. In general, this validation divides the total data set into ten subsets and tests each subset with the model trained by the remaining 9. Table 5.6 presents the evaluation results against different compilers and optimizations. The precision and recall rates represent the average of the ten tests. On average, FID has 0.916 precision, 0.959 recall and 0.930 F1 score for the 1,272 test cases. While the precision rate of binaries compiled from the `LLVM` compiler is slightly lower than the other two, the overall data is convincing. Indeed, most of the evaluation criteria on `gcc` and `icc` compiled binaries are quite *stable* to around 96% (besides the recall rate of `icc` O3). We interpret this as a promising result to show the semantics-based technique implemented in FID has good performance against various compilers and optimization levels.

We compare FID with existing research and industry de facto tools, which features handwritten patterns or data mining based function recognition. We also present the "baseline" method, i.e., only leveraging *call instruction collection* to recognize functions (Section 5.2.7). We evaluate all of them using the same test cases. Table 5.7 presents the average performance results; FID notably outperforms the baseline method, IDA-Pro and BW-Byte. Indeed by comparing with the baseline method, we have shown how FID can effectively improve the performance through the data mining-based method. BW-Instr can marginally outperform FID on normal code with no obfuscation; later we will see how the semantics-based model implemented in FID can surpass BW-Instr on obfuscated code. Note that this evaluation *only* tests the learned model, and in practice, *call instruction collection* (Section 5.2.7) can always provide additional information to improve the performance of FID.

Table 5.7: Comparison with different tools (the "baseline" method only uses *call instruction collection* to recognize functions).

	Precision	**Recall**	**F1 Score**
Baseline	1.000	0.527	0.690
IDA-Pro	0.998	0.600	0.750
BW-Byte	0.788	0.954	0.863
BW-Instr	0.996	0.997	0.996
FID	0.916	0.959	0.930

5.3.2 Different Compilers

In this section, we present the evaluation on distinguishing different compilers. As aforementioned (Section 5.2.6), for any given input binary, we aim to develop a classifier which can answer whether this binary is compiled by `LLVM` or `gcc/icc`.

Table 5.8 presents the performance of ten-fold validation against all the benign code. Most of the binaries compiled by `gcc/icc` can be correctly classified, with small errors (over 0.97 precision rate). As for the binaries compiled by `LLVM`, we report the recall rate is slightly lower than the other group. In particular, our finding shows that binaries compiled by `LLVM` and optimization O1 have similar distributions with binaries compiled by `gcc/icc` to certain degree. Nevertheless, given 1.000 precision and over 0.85 recall rate, we still interpret it as a promising result to show FID can recognize which compiler the input binary is compiled from for most of the cases.

Our tentative evaluation shows that training using data from binaries compiled by O0 and O2 optimization level can lead to stable performance, so to evaluate the obfuscation-resilience, we train the model using data from Coreutils binaries compiled by `LLVM` and `icc` compilers with O0 and O2 optimization levels (in total 424 program binaries),[5] and test the trained model towards all the obfuscated binary code (seven obfuscation methods and four optimization levels). Table 5.9 presents the average performance regarding different optimizations. We report besides four types of obfuscated binaries compiled by `LLVM` and O1 optimization, most of the obfuscated binaries perform flawlessly in this evaluation. This is consistent with our evaluation in Table 5.8.

[5]Considering the similarity of binaries compiled by `icc` and `gcc`, we choose to only use binaries compiled by one of them. Our test reports similar test results when substituting with `gcc` compiled binaries.

Table 5.8: Ten-fold validation on distinguishing different compilers.

Opt. Level	gcc/icc		
	Precision	**Recall**	**F1 Score**
O0	1.000	1.000	1.000
O1	1.000	1.000	1.000
O2	0.977	1.000	0.989
O3	0.940	1.000	0.969
Average	0.979	1.000	0.989
Opt. Level	**LLVM**		
	Precision	**Recall**	**F1 Score**
O0	1.000	1.000	1.000
O1	1.000	0.775	0.873
O2	1.000	0.84	0.913
O3	1.000	0.84	0.913
Average	1.000	0.863	0.926

Table 5.9: Evaluation on distinguishing different compilers on obfuscated binary code.

	Precision	**Recall**	**F1 Score**
O0	1.000	1.000	1.000
O1	1.000	0.391	0.488
O2	1.000	1.000	1.000
O3	1.000	1.000	1.000
Average	1.000	0.848	0.918

It is always possible to improve the recall rate when scarifying some precision. Besides, some tricks such as analyzing the exported function name and mangling schemes can also provide us with more clues. We leave it as one future work to improve the recall rate of binaries compiled by `LLVM` and O1 optimization. Overall, we assume FID can clearly distinguish whether an input binary is compiled by `LLVM` compilers or `gcc/icc` for most of the cases.

5.3.3 Obfuscated Code

The next step is to evaluate FID against the obfuscated binary code. In this test, we train a recognition model with normal binary code, and test the trained model with obfuscated code. Apparently, this is how FID is supposed to work in practice. Note that as obfuscated binaries are all compiled by `LLVM` compiler, as

Table 5.10: Evaluation on obfuscated code compiled with O3 optimization level. FID-CIC (5.2.7) outperforms all the other tools in terms of F1 score (i.e., the harmonic mean of *precision* and *recall*), which demonstrates the resilience of our technique towards obfuscated code.

Obf.	IDA-Pro			BW-Byte			BW-Instr			FID			FID-CIC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ins	1.000	0.474	0.643	0.719	0.929	0.811	0.911	0.920	0.915	0.958	0.683	0.798	0.966	0.839	0.898
opq	1.000	0.551	0.710	0.540	0.937	0.685	0.811	0.859	0.834	0.959	0.655	0.778	0.969	0.867	0.915
flt	1.000	0.489	0.656	0.730	0.924	0.816	0.716	0.936	0.811	0.913	0.607	0.729	0.933	0.806	0.865
mix1	1.000	0.543	0.704	0.569	0.931	0.706	0.768	0.907	0.832	0.955	0.586	0.726	0.968	0.840	0.899
mix2	1.000	0.489	0.657	0.494	0.935	0.647	0.685	0.915	0.783	0.896	0.621	0.733	0.918	0.809	0.860
mix3	1.000	0.560	0.718	0.395	0.929	0.554	0.671	0.942	0.784	0.943	0.608	0.740	0.958	0.835	0.892
mix4	1.000	0.565	0.722	0.444	0.925	0.600	0.703	0.941	0.805	0.849	0.557	0.672	0.895	0.842	0.868
Average	1.000	0.524	0.688	0.556	0.930	0.696	0.752	0.917	0.826	0.925	0.617	0.740	0.944	0.834	0.886

Table 5.11: Average performance evaluation on obfuscated code.

Opt. Level	FID		
	Precision	**Recall**	**F1 Score**
O0	0.979	0.852	0.911
O1	0.900	0.555	0.685
O2	0.944	0.616	0.745
O3	0.925	0.617	0.740
Average	0.937	0.660	0.774
Opt. Level	**FID-CIC**		
	Precision	**Recall**	**F1 Score**
O0	0.981	0.936	0.958
O1	0.933	0.885	0.907
O2	0.957	0.833	0.891
O3	0.944	0.834	0.885
Average	0.954	0.872	0.911

discussed in Section 5.2.6, we train FID with normal binaries compiled by `LLVM`. To be consistent with Section 5.3.2, we train FID with normal binaries compiled by optimization level O0 and O2 (in total 212 binary code). We also train BW-Instr and BW-Byte with the same settings.

We first report detailed results regarding the most challenging setting, that is, obfuscations with optimization O3. Table 5.10 reports the performance data. Note that BW-Instr also provides the functionality to improve the learned model with *call instruction collection* (Section 5.2.7), therefore to present a fair comparison, BW-Instr is configured with this functionality (column four). We do not configure BW-Byte with *call instruction collection*, as the high recall rate of BW-Byte shows small space for improvement (as aforementioned, "recall" rate represents the

percentage of real function entry points that is identified). As can be expected, FID-CIC outperforms all the other tools in terms of average F1 score, which is a strong evident to prove FID is resilient to obfuscated code.

In particular, our evaluation shows that while precision of FID is high whether we leverage *call instruction collection* or not, recall increases when it is applied (comparing column five and six in Table 5.10). This is consistent with our assumption, i.e., we conservatively select semantics features to guarantee precise recognition (low false positive rate), and eliminate false negative (improve recall rate) through *call instruction collection*.

Although BW-Instr shows better recall compared with FID, FID outperforms BW-Instr in terms of precision and F1 score. Thus, we interpret FID can have much lower false positive in analyzing obfuscated code. Again, we emphasize the recall rate can be *improved* by analyzing indirect function calls through value-set analysis, which is left as our future work.

We also reported the average performance score against all four optimization levels. As shown in Table 5.11, the average precision is over 0.95, and recall is over 0.87. Note that our evaluation shows that with more complex optimization applied, the recall rate decreases. We consider the main reason is that advanced optimization techniques are likely to decrease the need to use stack for parameter passing and locate variable allocation. In general, given overall 0.91 F1 score, we interpret that FID is capable of identifying functions even in front of binary code obfuscated by widely-used techniques with various optimization levels.

5.3.4 Execution Time

Our experiments are launched on a server machine with Intel Xeon(R) E5-2690 CPU, 2.90GHz and 128GB memory. In this section we report the time consumption of FID.

Feature Generation. FID takes 123.2 CPU hours to process all the normal binary code, and 1659.9 CPU hours for all the obfuscated code; the process time of each test case is recorded from starting to disassemble until finishing producing the numeric feature vectors. Naturally, as obfuscation complicates control flow structure and generates more basic blocks, it is reasonable to take more time to process.

Our study shows that there are two tasks taking more time than others, i.e., symbolic execution (Section 5.2.2) and tree edit distance computation (Section 5.2.4). On the other hand, while it takes a relatively long *CPU* time to process, as FID boosts several tasks (e.g., symbolic execution) with multithreading facilities, the *real* execution time is indeed much shorter. We report FID takes 23.1 real hours to process normal binary code and 483.8 real hours for obfuscated binary code. On average, it takes *7.2 minutes* to analyze one binary code.

Model Training. Our ten-fold validation training takes 3.62 CPU hour (Section 5.3.1), and the training time with only `LLVM` O0 and O2 binary code (Section 5.3.3) takes 178.1 CPU seconds. We consider the training time is in general promising.

Predication. Given the trained model and numeric feature vectors extracted from an input binary, predication is straightforward. We report that the predication time is 679.53 CPU seconds (Section 5.3.3). That is, on average it takes about 0.23s to recognize functions in one binary.

Chapter 6
Composite Software Diversification

Many techniques of software vulnerability exploitation rely on deep and comprehensive analysis of vulnerable program binaries. By transforming software into different forms before deployment, software diversification is considered as an effective mitigation of attacks originated from malicious binary analyses. Enlightened by research in other areas, we seek to apply different diversification transformations to the same program for a synergy effect such that the resulting hybrid transformations can have boosted diversification effects with modest cost. We name this approach the *composite software diversification*. In this chapter, we undertake an in-depth study in this direction and develop a reasonably well working selection strategy to find a transformation composition that performs better than any single transformation used in the composition. We start by demonstrating composite diversification with a motivating example and presenting its formal definition in Section 6.1. We then introduce how we setup our study in Section 6.2. Afterward, Section 6.3 details our selection strategy and Section 6.4 validates the design of composite diversification with more experiments.[1]

6.1 Introduction

With the rapid development of software reverse engineering and analysis, attackers have gained a certain level of advantages in the arms race. Code-reuse attack

[1]The work of this chapter is published in the 33rd IEEE International Conference on Software Maintenance and Evolution [101].

analyzes the victim programs to identify sequences of reusable code snippets and direct the control flow through these snippets to construct malicious operations [15, 102, 103]. Patch-based exploitation analyzes the post-patch binary code to expose hidden vulnerabilities (fixed by the patch) and construct attacks towards the pre-patch binary [104].

Software diversification produces different variants of a program, altering software syntax but retaining the semantic equivalence. After diversification, each copy of the software has a different structure. Therefore, knowledge obtained by reverse engineering one copy of the software is not applicable to other copies, making attacks depending on such knowledge (e.g., code-reuse attack and patch-based exploitation) lose generality or not feasible at all.

There has been many great work on software diversification and a large portion of them focus on the transformation algorithm side [48–50, 53, 105]. A good transformation algorithm can vastly mutate the binary form of a program with a considerable amount of randomness. Meanwhile, the transformation preserves the original semantics and keeps the incurred cost as modest as possible. Typical penalties of diversification transformations include binary size expansion and execution slowdown. As more and more transformation algorithms have been proposed, it becomes more and more difficult to develop new algorithms that provide reliable diversification effects with low cost. We have noticed that recent progress on software diversification is more about building frameworks and providing support for upstream techniques (binary rewriting, for instance) which enable software diversification in different scenarios [17, 19].

In this research, we propose *composite software diversification* which combines existing methods together for a synergy effect. The basic idea is that by applying different diversification transformations to the same program, we can make the binary more efficiently diversified compared to applying a single transformation; meanwhile, the cost of the composite transformation is kept low enough for practical deployment. The composite diversification, if feasible, can extract the hidden value of past research results and greatly enrich the choices of software diversification algorithms.

The idea of combining different program transformations of the same kind for greater benefits is not merely an intuition but has been proven reasonable and feasible by previous work on compiler optimization [106]. Although optimization

and diversification have different goals and are evaluated with entirely different metrics, we do believe that the success of the idea in one area gives a strong hint that similar methods can work in another field.

Given a set of primitive program transformation algorithms, the search space for an optimal or a close-to-optimal composition is considerably large. To investigate the feasibility and effectiveness of composite software diversification, we propose a methodology that comprehensively evaluates a diversification transformation, either primitive or composite. With this methodology, we further develop a strategy to prune the search space so that our study can be done in an empirical way; this strategy itself has a reference to the data mining research. To the best of our knowledge, despite the growing need for deploying diversified real-world applications, no systematic study has focused on comprehensively evaluating the performance of software diversification transformations when they are composed together.

In summary, we make the following contributions:

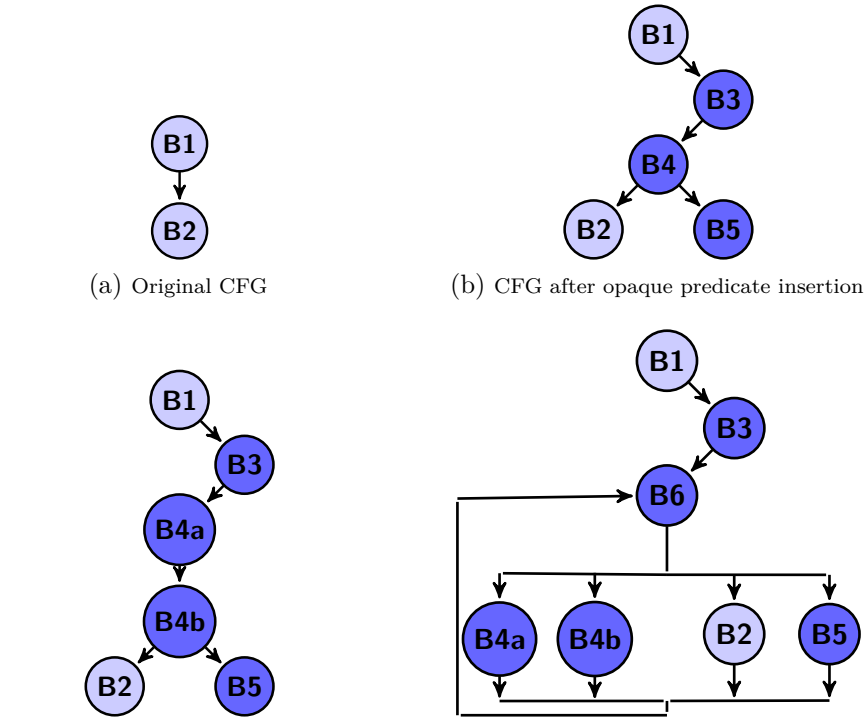
- We propose a new concept of software diversification called *composite software diversification*. By composing different diversification transformations in a certain way we can boost the effectiveness of the previously proposed methods while keeping the cost of diversification under control.

- By referring to data mining research, we develop a fairly well-performed search guideline on the basis of backward stepwise selection [107, 108]. This guideline effectively selects the satisfactory composition of transformations which diversifies a program without incurring much cost.[2]

- We justify our research idea and methodology with extensive empirical experiments on the C programs of the SPEC2006 benchmark suite. The results show that composite diversification is a promising technique that should be appealing to software developers and distributors.

- We developed a tool called Amoeba, which delivers composite software diversification to binary code (it is publicly available for download at `https:`

[2]Careful readers may notice there is no guarantee that stepwise selection can yield the globally optimal model; the proposed guideline performs a greedy search from composition candidates. On the other hand, the proposed guideline has been demonstrated as fairly well-performed in this research (details are given in Section 6.4). Hence we refer the constructed composition as "optimal" composition for simplicity purposes later in this thesis.



(a) Original CFG (b) CFG after opaque predicate insertion

(c) CFG after basic block splitting (d) CFG after control flow flattening

Figure 6.1: Program diversification in multiple iterations.

`//goo.gl/OYRLky`). To our best knowledge, there is no publicly available diversification tool on binary code, and we contribute to filling the gap.

6.1.1 Motivating Example

We now present a motivating example that demonstrates the synergy effect of composite software diversification. We then formalize the problem that our research is to solve in the next section.

The key observation is that a program can become more *diversified*, by composing multiple transformations on the processed binaries. We use a simple two block control flow graph to illustrate our observation. As shown in Figure 6.1a, the two blocks are connected by a direct `jmp` instruction. The original control structure is firstly transformed through an opaque predicate [62]. As shown in Figure 6.1b, `B1` will have to invoke function `B3` to get the predicate. A conditional `jmp` is implemented in `B4` to check the predicate, and if it fails, `hlt` instruction will be executed in `B5`. The `B4` is then transformed by splitting itself into `B4a` and `B4b` via

a inserted `jmp` instruction, as shown in Figure 6.1c. Finally, the output CFG is transformed by `control flow flatten`. As shown in Figure 6.1d, intra-procedural control flow graph is flattened (the graph in Figure 6.1d is simplified). Instructions are inserted to update each control flow transfer destination to a global variable, and the control transfers are redirected to `B6`. An indirect `jmp` in `B6` uses the address in the global variable to transfer the execution flow. Note that after three transformations, the 2-block graph is extended to a 7-block graph with a more complex structure. In summary, by reprocessing the output with different transformations, the diversified code can become progressively complex.

6.1.2 Problem

A program transformation could be formalized as a function $T : \mathcal{P} \to \mathcal{P}$ where $\mathcal{P}$ is the universe of all programs. In software diversification, T may be probabilistic, meaning the output of applying T to the same input is not unique but follows a distribution specific to T. In the rest of the text, we always consider diversification transformations.

Given two transformations T_1 and T_2, a mixture of them could be the composition $T_2 \circ T_1$, namely we first apply T_1 to a program and then apply T_2 to the output of T_1. Just like the expectation we have for any hybrid method, it would be ideal if $T_2 \circ T_1$ outperforms both T_1 and T_2 applied alone, concerning the given criterion for measuring the performance of software diversification. If $T_2 \circ T_1$ indeed has the better performance, we say there is a synergy effect between T_1 and T_2. The same study can be done with $T_1 \circ T_2$ which is another way to compose T_1 and T_2.

In this research, we would like to investigate if such synergy effect generally exists, especially when there are more than two primitive transformations to compose. If the synergy exists indeed, we want to develop a method that can effectively and efficiently achieve it. Since the synergy may manifest only if the transformations are composed in a particular way, this problem grows beyond trivial as the number of applicable primitive transformations increases. Therefore, the first step is to make a clear definition of the problem by generalizing the previously made example.

Suppose we have a set $\mathcal{T}$ containing k diversification transformations $T_1, \cdots, T_k$, then we can construct the set of all possible compositions in the following way.

$$
\begin{aligned}
C_n &= \{T_{i_n} \circ \cdots \circ T_{i_1} \mid \forall l \in \{1, \cdots, n\}, T_{i_l} \in \mathcal{T}\} \\
C^* &= \bigcup_{n=1}^{\infty} C_n
\end{aligned}
$$

Therefore, the objective of our research is to develop a search strategy which can find a subset of C^* such that the composed diversification transformations in this subset have the optimal or close-to-optimal performance under certain evaluation criteria.

6.2 Experiment Setup

Since there are very few mature formal theories on program diversification, we try to solve the problem in an empirical way. That is, by actually implementing a set of composite diversification transformations and developing a set of measurements we compare the performance of different compositions with field experiments. We believe that with carefully designed experiments and evaluation metrics, empirical results can still have a certain level of generality even without strong theoretical foundations.

From the construction of the problem space it can be seen that the choice of three factors decides a composition, which are 1) the length of the composition ("length" refers to the number of iterations), 2) the subset of primitive transformations to use, and 3) the order in which these transformations are composed. Apparently, enumerating all choices for the three factors is not feasible, so the major challenge of this research is how to reasonably prune the problem space so that the empirical evaluation can be done with limited resources.

In this research, we focus on the first two factors. Given the choices of the first two factors, we randomly decide the third factor to construct a composition. There are two reasons behind this decision. The first is that among all three factors, the third one expands the problem space most vastly. The second reason is that randomizing the sequence of composition potentially makes the diversification more unpredictable, which is a significant benefit in practice. Nevertheless, heuristic-based approaches, or perhaps machine learning-based approaches could facilitate the study of the third factor effectively. We leave it as one future work to investigate the third factor in boosting composite diversification.

Note that software diversification can be performed at different stages, e.g., at the pre-compile time, compile time, or at the post-link time (i.e., binary retrofitting). Although composite diversification in general is applicable at any of these stages, we choose to implement AMOEBA and demonstrate it with post-link time diversification because security hardening through binary retrofitting does not require source code and therefore has a broader application scope. We leave it to software developers to decide at what stage to perform composite diversification in the real-world scenarios.

The underlying reverse engineering facility AMOEBA relies on is an open-source disassembler called UROBOROS [40, 75]. Starting from the assembly code of the original binary, we iteratively apply different diversification transformations to the program, producing a unique variant each time. All transformations we use in UROBOROS are from existing software diversification research that are relatively *simple* and *straightforward*. The overlay of different simple transformations eventually leads to a synergy effect after a particular number of iterations, making the produced variants resilient to certain security threats and binary similarity detection.

6.2.1 Diversification Passes

We intuitively choose ten "classic" binary diversification methods proposed by existing research as the transformation candidates. In the rest of the paper, we name each transformation as a *diversification pass* applied to input programs. Most of these transformations have also been indexed by an influencing literature review on software diversification [50]. The roster of these transformations is in Table 6.1. Note that we use these simple transformations in our study, but more advanced and sophisticated transformations can be implemented as well. There are three levels of assembly transformations—instruction level, basic block level, and function level. At this point, it is still unknown whether each pass in Table 6.1 will be used in composite diversification or not. An in-depth selection process is required to decide an appropriate combination of passes for effective diversification. Pass selection will be discussed later in Section 6.3. We now elaborate on each pass.

Basic block reorder. This diversification rearranges the relative positions of basic blocks. In particular, two basic blocks are randomly selected as candidates

Table 6.1: Diversification pass candidates.

Class	Methods
Instruction Level	instruction replace [48] instruction insert [48]
Basic Block Level	basic block reorder [49] basic block merge [50] basic block split [50] opaque predicate insert [62] control flow flatten [62] branch function insert [105]
Function Level	function reorder [53] function inline [50]

to reorder, with necessary control transfer instructions and labels inserted in the context to preserve the correct semantics. For each pass, we reorder one pair of basic blocks from each function.

Basic block split. This diversification splits one basic block into two by inserting a `jmp` instruction in an arbitrary position and set its destination to the associated next instruction. We randomly select one basic block to transform from each function.

Basic block merge. This transformation searches for mergeable basic blocks. A basic block is defined as mergeable if it has only one predecessor, and its predecessor has only one successor. For each pass, we randomly select one pair from all the mergeable basic block pairs and merge them together.

Instruction insert. This diversification inserts meaningless code sequences into the program. For each function, we randomly select one basic block and insert a sequence of garbage instructions with a random length (3–5). The insertion candidates are `nop`, `mov %esp, %esp`, `lea 0x0(%esi),%esi`, and `xchg %esp, %esp`.

Instruction replace. This strategy searches for typical instructions and replaces the targets with its semantic equivalent substitutions. We adopt two substitution strategies to transform `call` and `ret` instructions. When replacing `call` instructions, `jmp` instructions are used to transfer the control, and the return address is explicitly saved on the stack by a `push`. `Ret` instructions are replaced in a similar way, with a `pop` instruction to obtain the return address on the stack.

Control flow flatten. As shown in Figure 6.1d, this transformation flattens the control flow graph. Given a target control structure, control flow transfers are redirected to a dispatcher block inserted by this transformation. The dispatcher leverages a global variable to decide which block to jump, and instructions are inserted at the end of each basic block to update the global variable with control destinations. For each pass, we randomly select one function and flatten its control flow graph for each iteration.

Opaque predicate insert. As shown in Figure 6.1c, a basic block `B` can be guarded with a conditional branch to `B` and another (garbage code) block `B'` using an arbitrary predicate. A `call` instruction to the predicate function and a conditional jump instruction are inserted at the beginning of target block. For each pass, we insert one opaque predicate for each function.

Branch function insert. This transformation substitutes `jmp` instructions with `call` instructions to the branch routine and the `jmp` destinations are updated into an artificial global variable. The branch function utilizes an indirect `jmp`, transferring to the destination stored in a global variable. We transform all the identified candidates.

Function inline. This transformation inlines functions into their call-sites. Direct `call` instructions to the target function are found and the target function is inserted after these call-sites. The `call` instructions are changed into `push` and `jmp` instructions to simulate the original semantics. `Ret` instructions in the target are also rewritten into `jmp` instructions. As destinations of an indirect function call are hard to analyze, we conservatively leave the target function in its original place. In the implementation, one function is randomly selected to transform as long as its size is less than a predefined threshold (the threshold is set as 500 bytes in this paper).

Function reorder. Same as basic block reorder, this transformation rearranges the relative positions of two functions. In case the execution flow falls through the function boundaries, we insert `jmp` instructions and corresponding labels in the predecessors and successors of the reordered pair, thus delivering the equivalent semantics. For each pass, we reorder one pair of functions.

6.2.2 Measurement

The goal of composite diversification is to provide production of low cost and well-diversified software variants. To assess our fulfillment of this objective, we need to quantitatively measure cost and diversity.

6.2.2.1 Cost

We assess cost with two metrics—size expansion and execution slowdown of diversified binaries. The cost is a concern in composite diversification because most passes in Table 6.1 insert new instructions or new control flow transfers into the binary, which will inevitably affect the size and speed of the products. We leverage `bzip` (Section 6.3 and Section 6.4.1) and SPEC2006 programs (Section 6.4.2) in our experiments. The execution speed (i.e., slowdown) of diversified variants are measured through the test cases shipped with the programs. The size is calculated using the `stat` program from GNU Coreutils. Our experiments are launched on a machine with Intel E5-2690 2.90GHz with 128GB memory running Ubuntu 12.04 LTS.

6.2.2.2 Diversity

Another aspect of the assessment is to measure the diversity of the variants produced by composite diversification. We present our quantitative evaluations on diversity regarding two typical threats that software diversification can hinder, i.e., code reuse attack [15] and patch-based exploitation [104]. The security strength of a diversification method can be well reflected by its resilience to these two adversaries.

Resilience to code reuse attacks can be evaluated by the elimination rate of return-oriented programming (ROP) gadgets. ROP attack is one state-of-the-art program exploitation which manipulates program call stacks and chains sequences of victim program's own code snippets (named ROP gadgets) to perform arbitrary operations [15, 16]. A general assumption made by related work is that attackers need to know the memory addresses of ROP gadgets in order to tamper the call stack [17, 19, 50, 109, 110], and if a ROP gadget changes its location or no longer exists in the diversified binary, attackers will have difficulty in reusing the existing attack payloads. We use the ROP gadget harvesting tool ROPGadget [74] to search

for gadgets in binaries. Gadget elimination rate between two binaries is defined as

$$1 - \frac{|A \cap B|}{\min\{|A|, |B|\}}$$

where A and B are the sets of gadgets found in two binaries. Two gadgets are considered equal if they have the same instruction sequence and starting address. Note that recent research work has proposed advanced methods to launch ROP attacks without the pre-knowledge of ROP gadgets (i.e., Just-In-Time ROP attack [111, 112]).

Resilience to patch-based exploitation can be evaluated by investigating how well the diversified binaries can mislead binary diffing tools [61], which can be used to locate the vulnerability by comparing the semantics of the original binary and the patched one. We use `BinDiff` (version 4.0.1) [86], the de facto industrial standard binary diffing tool available on the market, to calculate the similarity between two binaries. Given two binaries, `BinDiff` provides the number of matched functions, basic blocks, and instructions. Since function and basic block can be "partially" matched, e.g., 30% of the instructions in a function are matched with another function, counting the number of matched functions or basic blocks could be tricky. Therefore, we only adopt instruction matching rate. The rate is calculated by

$$\frac{|A \cap B|}{\min\{|A|, |B|\}}$$

where A and B are the sets of instructions in two binaries. Two instructions are considered equal if they are matched by `BinDiff`, so $|A \cap B|$ is the number of matched instructions. It should be noted that being semantic equivalent does not necessarily make two instructions a match; `BinDiff` also takes the contexts of the instructions into account [113,114]. `BinDiff` does provide an overall similarity score to summarize the comparison. However, it is unclear how this score is computed. To make our results more interpretable, we do not use it in our evaluation.

Also, unlike cost assessment which only compares every diversified binary with the original one, evaluation on the diversity of composite diversification needs an additional step. Since attackers are usually not limited to only chose the original binary to analyze, the diversity of variants should reach the **pairwise** granularity.

That means, every pair of the generated variants should be different enough so that attackers cannot exploit any other variant by reverse engineering one of them.

6.3 Pass Selection

Given the candidate passes in Table 6.1, we want to find an applicable subset of them as the primitive transformations to employ in composite diversification.

6.3.1 Selection Methodology

Having decided the metrics used for assessing the performance of composite diversification, we can start searching for the subset of diversification passes that can be employed by our implementation of composite diversification. Given 10 passes, there are a total of 1023 different non-empty combinations of them if we do not fix the number of passes to pick. Assessing all possible combinations is unlikely to be feasible.

To address this issue, we propose a two-stage pass selection method. In the first stage, we evaluate the cost of every single pass when they are repeatedly applied to a binary for many times. After this first-stage selection, passes that are too costly in the context of composite diversification will be ruled out for further consideration. Hopefully, the first-stage selection can reduce the total number of passes we need to consider in the second stage.

We leverage program `bzip2` (version 1.0.3), a widely-used data compressor as the experiment object in the selection process. For each diversification combination, we iterate it for 500 times, which we believe is significant enough for an in-depth study. These 500 iterations lead to 500 variants, each of which is based on the previous one instead of the the original. Before launching selection steps below, we first verify the functional correctness of these diversified outputs using the test cases shipped with `bzip2`. We report all the outputs can pass these shipped test cases.While the adopted algorithms are supposed to produce equivalent code, we test the functional correctness to confirm the faithful implementations of these algorithms in Amoeba.

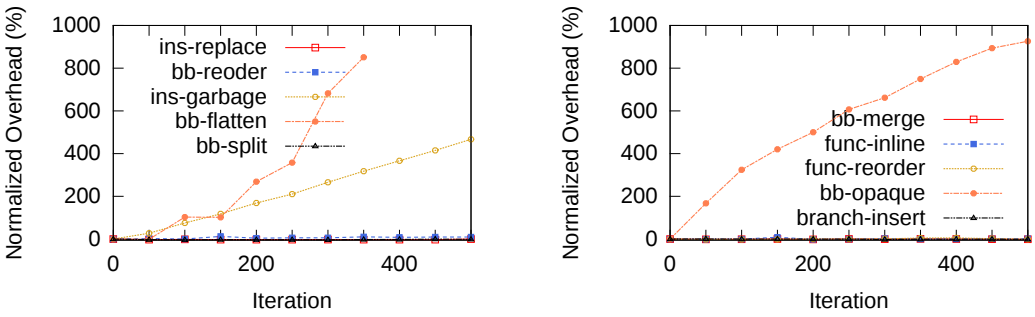


Figure 6.2: Execution slowdown by single-pass diversification.

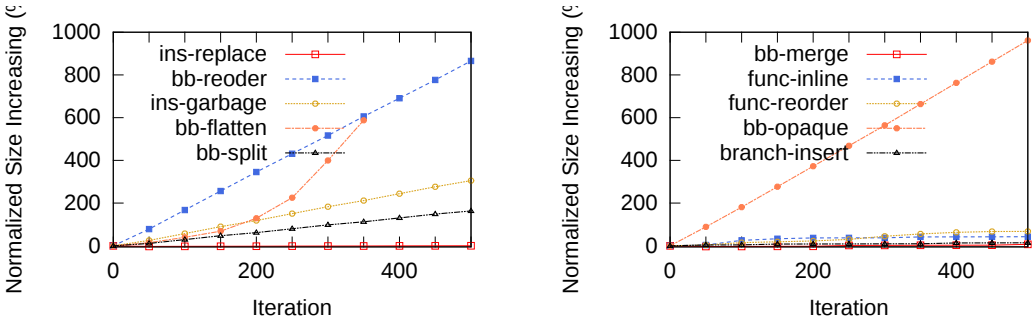


Figure 6.3: Size expansion by single-pass diversification.

When comparing the diversified binaries with the original one, we do a 1-in-50 sampling on the 500 variants, leading to a sample size of 10. For pairwise comparison on diversity-related metrics, we randomly pick 50 pairs of variants.

6.3.2 First-Stage Selection

The first-stage selection measures the singular cost of each diversification pass. We consider a pass to be too costly if the size expansion grows super-linearly, or the execution slowdown does not grow sub-linearly with respect to iteration times because users are usually much more sensitive to execution speed than binary size increases.

Figure 6.2 and Figure 6.3 show the size expansion and execution slowdown of diversified `bzip2` variants over 500 iterations, for all 10 diversification passes. While most of the transformations only introduce negligible runtime overhead, the impact of basic block flatten, instruction insert, and opaque predicate insert is out of the scope of our tolerance for execution slowdown. As for size expansion, the increasing trends of most passes are linear, except basic block flatten. According

Table 6.2: Candidate diversification pass combinations generated by backward-stepwise selection (mix2 indicates the best of all).

	mix0	mix1	mix2	mix3	mix4	mix5
Instruction replace	✓	✓	✓	✓		
Basic block reorder	✓	✓	✓	✓	✓	✓
Basic block merge	✓	✓	✓	✓	✓	
Basic block split	✓	✓	✓	✓	✓	✓
Branch function insert	✓					
Function reorder	✓	✓	✓			
Function inline	✓	✓				

to our definition of costly transformations, basic block flatten, instruction insert, and opaque predicate insert will be excluded from further consideration by our composite diversification framework, reducing the number of candidate passes from 10 to 7.

6.3.3 Second-Stage Selection

Although the first-stage selection has pruned a few passes, the remaining search space is still too large for us to enumerate. Therefore, we need a strategy to further compress the pass selection process in the second-stage.

After referring to previous research and related disciplines, we decide to borrow a selection method from data mining. There is a classic problem in data mining called regression which seeks to estimate the relationship between a response variable and a set of predictor variables. In regression, a mathematical model with configurable parameters is assumed, from which the response can be computed based on the values of predictors. However, it is common that only a subset of the predictors are actually related to the response, so regression has to decide which predictors should be selected for fitting the model. Similar to our situation, enumerating all possible combinations of predictors usually needs unaffordable resources. Therefore, data mining researchers have developed numerous predictor selection methods to avoid brutal-force search for an optimal model.

In this research, we employ the *backward-stepwise* method [107, 108] for pass selection. In backward-stepwise selection, the baseline is first set as the model containing all candidates (predictors in data mining and diversification passes in our case). Starting from this baseline, the selection process generates a set of new

models by removing one candidate from the baseline model. Among all newly generated models, the one with the best performance is picked as the new baseline. The selection process repeats in this way until the latest baseline model consists of only one candidate. For our problem, we stop when the baseline model has only two passes, because we have already assessed the performance of every single pass. When the selection process is over, each baseline model is considered as the best model among all models with the same number of candidates. The final step would be comparing all baseline models.

There is a similar method called forward-stepwise selection. The difference is that instead of eliminating candidates from the baseline model one by one, forward-stepwise gradually adds new candidate starting from the empty model. Compared to backward selection, forward selection tends to generate a model with fewer candidates. Note that in composite diversification, we want to *maximize the diversity* of the passes we use in the framework, so we choose to build the model backward instead of forward.

When comparing two diversification plans for large iterations, it is hard to give a particular criterion that will be universally applicable. Depending on the characteristics of the program to protect and the demand of users, the comparison result could be different. We do not try to develop a versatile comparator for evaluating diversification plans. Instead, we only propose a reasonable comparator to illustrate the feasibility of our proposed framework.

In this work, we assume users care about execution slowdown most, so it will be the decisive factor when deciding the best diversification plan. On the other hand, we do not want a winning plan that suffers from some obvious drawbacks. For that purpose, we design the comparison method in a filter-oriented manner. A plan is discarded if it does too badly with respect to any one of the metrics mentioned in Section 6.2.2. After the filtering is over, we pick the best plan based on the metric we care about most, which is execution slowdown in our illustration.

A plan is considered to be too poor at a metric if its score on that metric is an outlier in the undesirable direction among the scores of all plans. In statistics, a data point is called an outlier if it is greater than $Q_3 + (Q_3 - Q_1) \times 1.5$ or smaller than $Q_1 - (Q_3 - Q_1) \times 1.5$, where Q_1 and Q_3 are the first and third quartile. For example, if the execution slowdown of some plan is an outlier at the high end, we will filter out that plan because high execution slowdown is unwanted. On the other

Table 6.3: Mean of metrics used for plan selection.

	mix1	mix2	mix3	mix4	mix5	Norm. Range
Slowdown (%)	3.42	-0.81	0.45	-0.31	3.20	[-5.59, 8.47]
Matching (%)	16.08	16.04	14.96	14.90	86.89	[13.28, 17.76]
Pair.Match. (%)	47.25	35.72	34.88	34.70	38.84	[28.94, 44.78]
Elim. (%)	98.21	98.21	98.22	98.22	98.22	[98.20, 98.24]
Pair.Elim. (%)	98.01	97.97	98.12	98.29	98.46	[97.59, 98.71]

hand, if the gadget elimination rate of some plan is an outlier at the low end, we will also filter it out because low gadget elimination rate indicates poor diversity.

Note that the filter-based selection may result in a situation where all candidate plans are pruned. That would mean every plan has at least one weakness. Any of such situations happening would threaten the rationality of our selection method. Nevertheless, none has manifested in our experiments.

Due to limited space, we are unable to show the analysis result of all combinations that have appeared in the selection process. We only present the final selection, i.e., selecting the best baseline model. Recall that the baseline models are the best-performing combinations in each round of backward-stepwise selection. We list these combinations in Table 6.2.

To illustrate the selection process, we first present the performance data of the diversification plans. We measure all metrics for mix1 to mix5. mix0 only has data reflecting execution slowdown. Since its runtime overhead is clearly unacceptable for composite diversification (up to 462% after 500 iterations), there is no need to consider it in subsequent selection.

Figure 6.4 shows the execution slowdown of each combination. As can be seen, the overhead of mix0 is significantly higher than the rest, while overheads of other plans are considered sub-linear (satisfying for further study). Figure 6.5 shows the size expansion evaluation. All the plans show roughly linear expansion, which is quite consistent with the size evaluation in the first-stage selection (Section 6.3.2). For diversity between the generated variants and the original binary, Figure 6.6 and Figure 6.7 show the ROP gadget elimination rate and instruction matching rate from `BinDiff`, respectively. As shown in Figure 6.6, almost all the ROP gadgets are eliminated after transformations; we report on average 98.2% gadgets become un-reusable. Binary diffing evaluation also shows promising results. Actually besides `mix5` (which will be filtered out due to the high remaining similarity), all

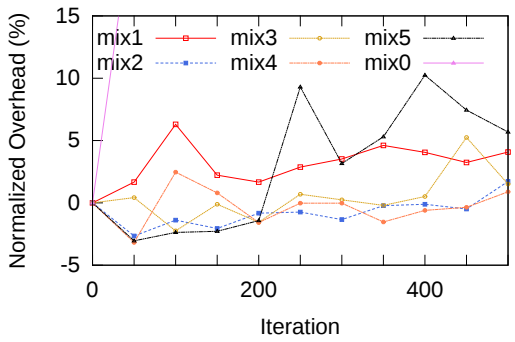


Figure 6.4: Execution slowdown by multi-pass diversification.

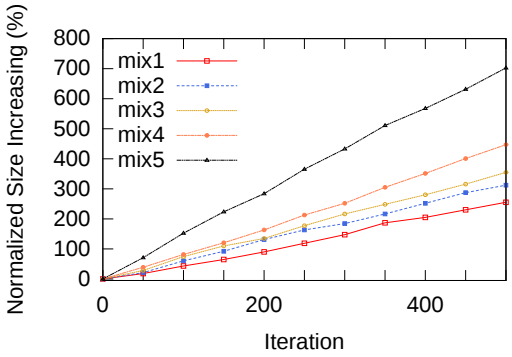


Figure 6.5: Size expansion by multi-pass diversification.

the plans show notable decrease in the instruction matching rate. We also present the pairwise diversity evaluation in Figure 6.8 and Figure 6.9. Pairwise ROP gadget elimination rate shows promising results; we observe stable high elimination rates for all the compared pairs. As for the pairwise instruction matching, we report `mix1` has relatively high remaining similarity (47.3%), while the other four plans show much better results (on average 36.0%).

Table 6.3 summarizes the experiment data by computing the average of each metric. As previously described, we pick the plan with the lowest runtime overhead, after filtering out plans with salient weaknesses. The normality range of each metric is computed, also listed in the table. As can be seen, mix1 is pruned due to lack of pairwise diversity; mix5 is filtered out because of high similarity between the variants and the original binary. mix2 is selected as the best diversification plan because it has the lowest execution slowdown and does not suffer from any significant weakness.

6.3.4 Multi-Pass Versus Single-Pass

After the two-stage pass selection, we have chosen mix2 as the winner combination of transformation passes for composite diversification. However, we have not yet showed that mix2 can outperform single-pass diversification in terms of diversity. To prove that the synergy effect we have discussed in Section 6.1.1 does exist, we launch a competition between multi-pass and single-pass diversification. Measurement on

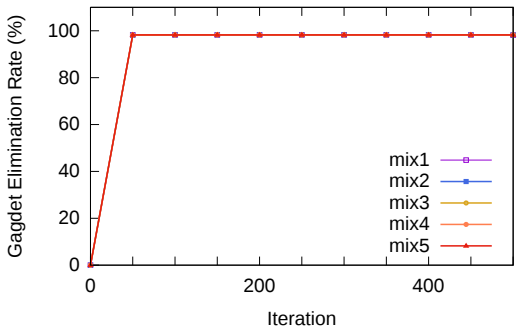


Figure 6.6: ROP gadget elimination by multi-pass diversification.

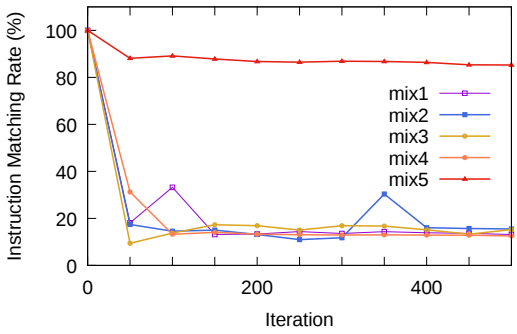


Figure 6.7: Binary diffing by multi-pass diversification.

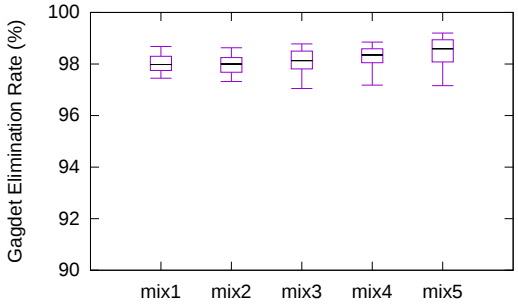


Figure 6.8: Pairwise ROP gadget elimination by multi-pass diversification.

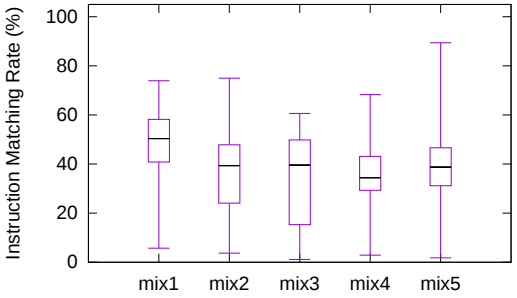


Figure 6.9: Pairwise binary diffing by multi-pass diversification.

the diversity of single-pass generated variants is illustrated in Figure 6.10, 6.11, 6.13, and 6.12.

We first make a comparison between single-pass diversification and multi-pass diversification on their resilience to ROP attacks. Figure 6.10 and Figure 6.6 suggests that there is no significant difference between single-pass and multi-pass plans on ROP gadget eliminate rates when comparing the variants with the original binary. In the pairwise comparison showed in Figure 6.12 and 6.8, however, the performance of some single-pass plans is clearly inferior to multi-pass plans. For example, instruction replace has nearly zero pairwise diversity. The reason is that after the first round of diversification, there are no suitable instructions for this

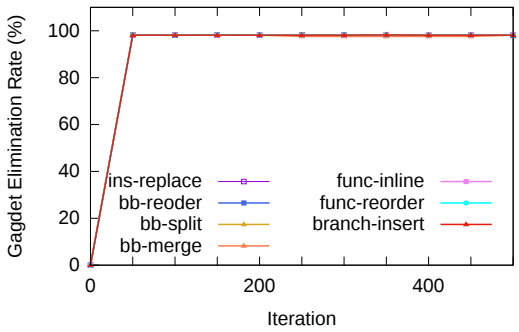


Figure 6.10: ROP gadget elimination by single-pass diversification.

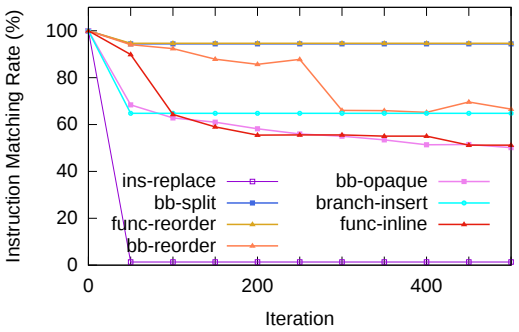


Figure 6.11: Binary diffing by single-pass diversification.

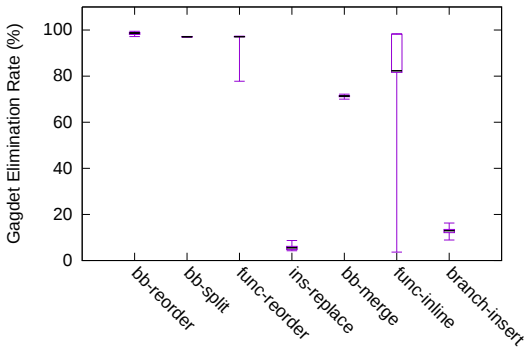


Figure 6.12: Pairwise ROP gadget eliminate by single-pass diversification.

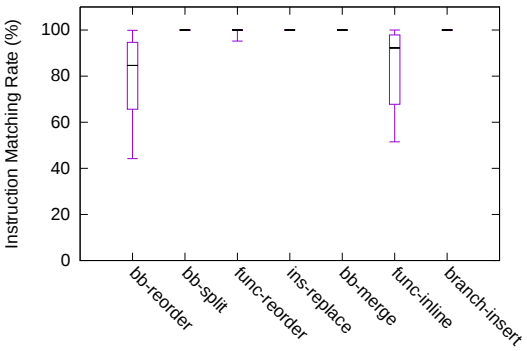


Figure 6.13: Pairwise binary diffing by single-pass diversification.

pass to replace, making it an idempotent transformation. Differently in multi-pass diversification, the collaborating passes (basic block reorder, for instance) keep inserting instructions that can be replaced, making instruction replace an efficient pass throughout the iteration.

In the competition on resilience to patch-based exploitation generation, the advantage of multi-pass diversification is even more significant. When matching the diversified binaries with the original (Figure 6.11 and 6.7), most multi-pass plans can reduce instruction matching rate to about 20% after 50 iterations. In contrast, matching scores from most single-pass plans stay above 45% even after 500 iterations.

The only exception is instruction replace, which can achieve nearly 0% matching score. Our guess is that replacing `call` and `ret` with `jmp` causes exceptional hardship for `BinDiff` when it tries to recover the control flow, which is the crux of its matching algorithm. Nevertheless, instruction replace is an idempotent pass, meaning it will surely have poor performance on pairwise matching. Actually, Figure 6.13 and 6.9 show that all multi-pass plans provide much more pairwise diversity than single-pass plans. At this point, we have enough evidence to conclude that multi-pass diversification is more effective than single-pass diversification. This conclusion further justifies the election of mix2.

6.4 Validation

In this section, we validate our approach in two aspects. We first compare our backward stepwise selection against a baseline approach, i.e., random selection (Section 6.4.1). We also validate our optimal combination with multiple large size programs from the SPEC2006 test set (Section 6.4.2).

6.4.1 Comparison with the Baseline Method

In our pass selection step (Section 6.3), we initialize our selection from ten widely-used program diversification methods. Since the ordering between different techniques are not considered (Section 6.2), ten methods lead to 1023 non-empty combinations. After the first phase, 3 of the methods are eliminated, resulting in 127 different possible combinations. Our tentative tests show that it takes a non-trivial time to apply transformations for hundreds of iterations. Thus, running diversification for all 127 combinations and pick the best available option cannot be done in a reasonable amount of time. As presented in Section 6.3.3, by using stepwise selection to find the optimal combination, we have to test 27 different candidates (7+6+5+4+3+2). In this section, we study whether a random selection approach (i.e., the *baseline method*) can find a better combination in 27 different runs.

We randomly select 27 combinations from 120 possible combinations which contain *at least two methods*. Our study on the `bzip2` program has shown that after 100 times, there is no significant benefit for most of the methods regarding

binary similarity (Figure 6.7). Thus, the process is iterated for *100* times for each combination. We evaluate the performance and cost of the 100th diversified output; we also compare the 100th and the 50th diversified outputs for the pairwise metrics (in this step, experiment results of mix2 are acquired in the same way). The optimal combination from 27 candidates is selected regarding the same filter-based selection strategy employed in Section 6.3.3.

We launch this 27-round random process for 20 times, resulting in 20 control groups.[3] We now report the key observations. In general, all 20 control groups show comparable execution slow and size expansion with mix2. On the other hand, while binaries from control groups has acceptable (pairwise) ROP gadget elimination rate and low similarity rate, we observe 18 control groups suffer from unsatisfying *pairwise* similarity rate. In particular, test on mix2 reports a low pairwise similarity rate (around 40%), while 18 control groups have over 95% pairwise similarity rate. There are only two well-performing control groups, i.e., group6 and group10. We report that group6 has the exact same combination with mix2, while group10 has a combination of three methods, i.e., instruction replacement, basic block reorder and basic block split. Overall, our study shows that for these 20 control groups, only two of them show comparable performance with mix2, and there is no control group can notably outperform mix2. We interpret this as promising results to show our stepwise selection can quickly construct optimal combinations with reasonable amount of effort.

6.4.2 Test on SPEC Programs

In Section 6.3 we have selected a set of diversification passes as the primitive transformations to use in composite diversification, based on experiment data on a single program `bzip2`. In this section, we validate our winning diversification plan mix2 on a larger set of programs, i.e., all C programs in SPEC2006. For validating the cost and effectiveness of the diversification plan, the experiment setting and metrics to measure are same as the selection process. Although our experiments on `bzip2` indicate that there may not be obvious benefit after 100 iterations, the SPEC programs are still conservatively processed for 500 iterations. We hope this *extreme* setting can better reveal the advantage and limitations of our technique

[3]The full comparison can be viewed at `https://goo.gl/3XbtVR`.

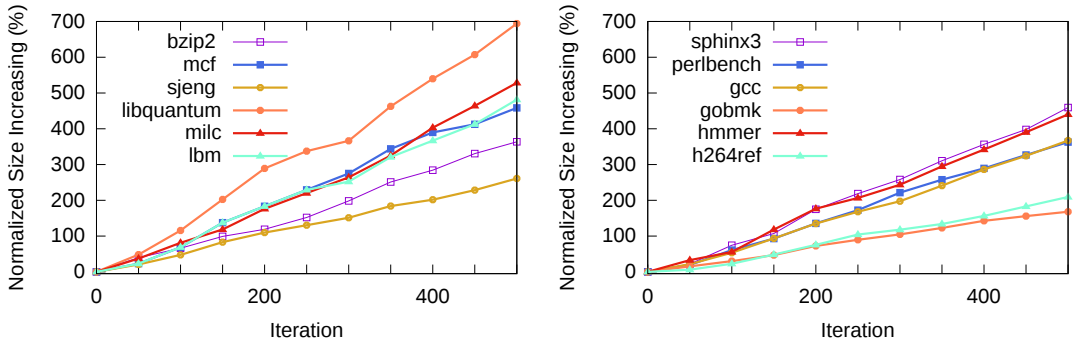


Figure 6.14: Size increase for SPEC binaries.

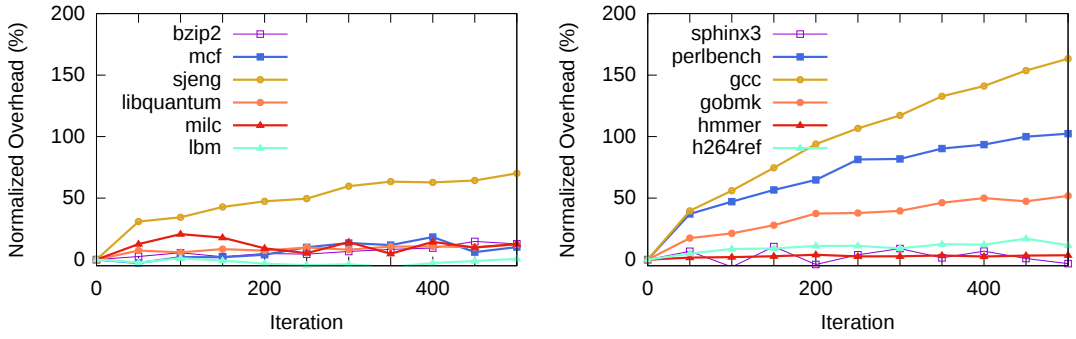


Figure 6.15: Runtime overhead for SPEC binaries.

and facilitate software developers with detailed information regarding real-world deployments, while a user can choose the needed number of iterations in practice. Same as Section 6.3.1, before launching experiments below, we first verify the functional correctness of all the diversified outputs. We report all the outputs can pass the test cases shipped in SPEC2006.

The size expansion of binaries used for validation is given in Figure 6.14. The data shows that the augment of binary size is bounded by 700% in 500 iterations, for all tested programs. Moreover, the trend of size expansion is linear with respect to original binary size and number of iterations. We believe this amount of cost is acceptable, at least for desktop and server computing environments.

The execution slowdown of the diversified binaries is probably more of a concern for composite diversification. The trend of runtime overhead increase over iterations is presented in Figure 6.15. According to the graph, overhead of all programs grows sublinearly, which fits our objective.

Table 6.4: Mean of performance metrics for C programs in SPEC2006.

	bzip2	mcf	sjeng	libquantum	milc	lbm	sphinx3	perlbench	gcc	gobmk	hmmer	h264ref	Mean
Slowdown (%)	7.16	7.50	52.52	9.05	12.13	-2.30	2.61	75.48	107.90	37.70	2.82	10.73	26.94
Matching (%)	32.66	24.88	1.60	10.49	11.63	23.58	5.89	13.73	6.56	7.09	9.85	6.18	12.84
Pair.Match. (%)	44.89	33.51	37.80	52.70	43.01	31.88	38.34	88.82	38.98	45.37	51.87	60.73	47.32
Elim. (%)	99.05	96.42	100.00	99.21	99.64	97.26	99.71	100.00	99.98	100.00	99.82	100.00	99.26
Pair.Elim. (%)	96.47	90.69	98.27	97.12	98.46	92.46	98.93	99.76	99.90	99.80	98.99	99.27	97.51

For diversity validation, Figure 6.16 presents evaluation on the ROP gadget elimination. Almost all the ROP gadgets become unavailable after transformation. We also report besides three test cases which have a relatively high instruction matching rate (`bzip2`, `lbm`, `mcf`), average matching rate of all the other test cases are less than 15% (Figure 6.17).

Figure 6.18 and 6.19 present the pairwise diversity. While all the pairwise gadget elimination tests show promising results (on average 97.51% gadgets are eliminated), we observe one outlier (`perlbench`) in the pairwise binary diffing evaluation. Its relatively large size of program code section is probably the main reason for the low diffing rate. On the other hand, we report the average diffing rate of other cases is 43.87%, which is promising.

Table 6.4 presents a summary of the performance data gathered from the validation process, showing the same set of metrics as we do pass selection in Section 6.3.3. While on most metrics, the validation result is consistent with the selection process, there may be some concern about execution slowdown. For 8 out of 12 programs, composite diversification introduces less than 15% runtime overhead on average. However, the impact on the other 4 programs is much more significant, leading to an average slowdown from 37.70% to 107.90%. Our observation is that, programs with more complicated control flows tend to be penalized more by our diversification plan. This is under intuition since many passes in mix2 focus on disturbing the control flow.

Apart from metrics that have been used for pass selection, the processing time needed for generating diversified copies is also an important factor affecting the deployment of our framework. We report on average, it takes 248.1 seconds to process a binary in one iteration of diversification. The time is measured on the same machine to evaluate the runtime overhead, whose specification is posted in Section 6.2.2. We also measure the relationship between the average processing time for one iteration and the original binary code size. We fit processing time with code size by a linear function with zero intercept. The regression test shows that

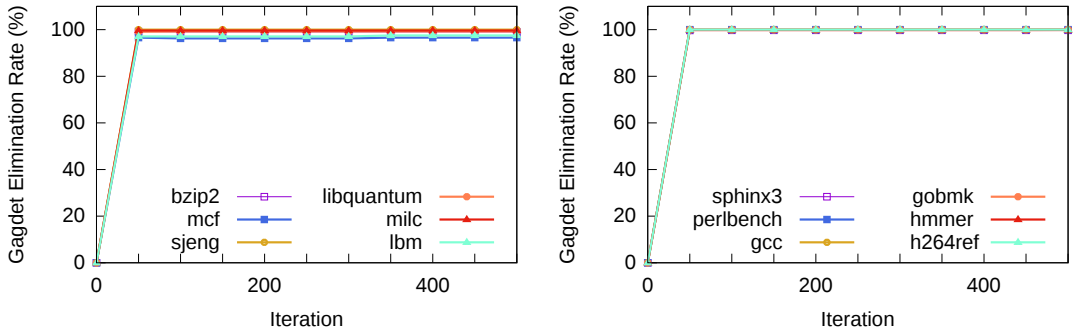


Figure 6.16: ROP gadget elimination for SPEC binaries.

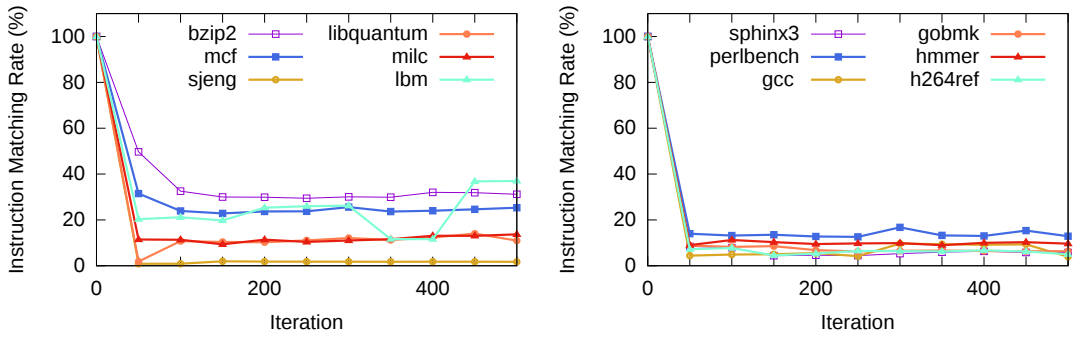


Figure 6.17: Binary diffing for SPEC binaries.

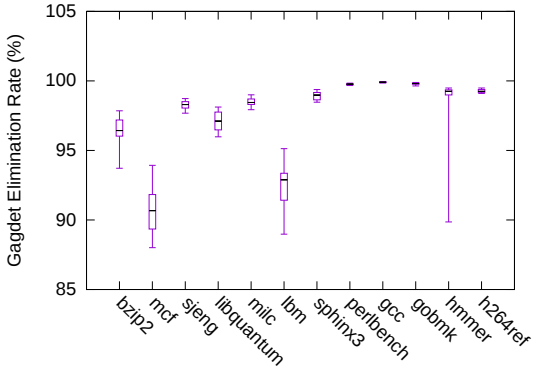


Figure 6.18: Pairwise ROP gadget elimination for SPEC binaries.

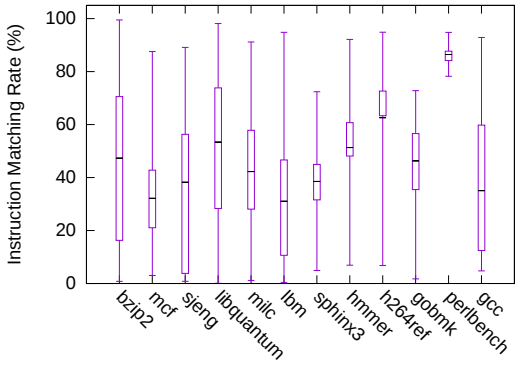


Figure 6.19: Pairwise binary diffing for SPEC binaries.

the linear relation (with the slope as 0.695) is significant at the confidence level of 99%. Note that the processing time is the average of an *extreme* setting with 500 iterations. That means, as the average processing time increases almost linearly regarding the code size, we can expect a notable decrease in processing time when binaries are diversified with a smaller number iterations in the real-world usage.

Chapter 7
Discussion and Future Work

We believe that we have built an enabling technology that could be employed as the basis of many important research and applications, such as software fault isolation (SFI), control-flow integrity (CFI), ROP defense, and in general software retrofitting for binary code, which is extremely important for legacy code systems. Nevertheless, this is a first step in the toolchain development. In this chapter we discuss the application scope of UROBOROS. We also discuss some short- and long-term directions that we see as promising to extend UROBOROS, such as supporting emerging platforms and bridging with the LLVM compiler framework.

7.1 Application Scope

UROBOROS is mainly designed to process stripped binaries without debug or relocation information. Most challenges originate from the difficulties in precisely disassembling binary program and producing relocatable code. As a result, we assume input binaries are not obfuscated. In addition, we assume binaries to instrument do not dynamically generate code or feature self-modifying.[1] Binaries compiled from typical C programs fit these assumptions very well.

Currently, UROBOROS mainly takes binaries compiled from C code as the input. C++ programs can be processed by UROBOROS as long as it does not rely on the exception handling. Since binaries store the exception handling meta-data as separate sections, parsing these sections requires additional engineering efforts. We present further discussion on C++ binary support soon in Section 7.3.

[1]Note that UROBOROS and other static tools face similar challenges on correctly disassembling obfuscated binary code or self-modifying code [17, 115].

By the time of writing, UROBOROS supports ELF binaries on both x86 and x64 architectures. UROBOROS makes no assumptions on what compilers are used to generate the input binaries (compiler compatibility is discussed soon in Section 7.2), and it produces relocatable assembly code regardless of the input binaries being stripped or not. We believe that UROBOROS has initiated a new focus on binary reverse engineering and instrumentation by delivering reassembly-based instrumentation.

7.2 Compiler Compatibility

Sometimes binary reverse engineering is compiler dependent, but UROBOROS does not explicitly depend on any compiler-specific features as far as we know. To roughly investigate UROBOROS's compatibility with other compilers, we try to disassemble and reassemble some binaries compiled by Clang, another widely used compiler different from GCC. We repeat the same functionality verification described in Section 3.5.1 on the 32-bit binaries in REAL, which are compiled by Clang this time. The applied assumption set in this experiment is the empty assumption set, and all reassembled binaries can pass the functionality tests. We plan to test UROBOROS's compatibility in more depth in the future.

7.3 C++ Binary Disassembly

The C++ programming language has more specific features compared with C. Binaries compiled from C++ programs often contain more sections to store meta-information. At this point UROBOROS still cannot fully support C++ disassembly, but we have already worked out a blueprint on how to recover these sections. There are two kinds of meta-information sections specific to C++. We now briefly discuss how to recover them.

- The `.ctors` and `.init_array` sections contain the addresses of constructor functions—functions that need to be executed at start up before the `main` function takes control. These sections can be directly dumped out and symbolized by treating them as data sections.

- The `.eh_frame` and `.gcc_except_table` sections store the information used for stack unwinding and exception handling for C++ programs in the DWARF format [116]. There have been some reverse engineering tools, e.g., Katana [117] and IDA Pro, that can parse the DWARF data. By understanding the semantics of a DWARF entry, we can adjust its content and make the reassembly flawless.

7.4 Support Emerging Platforms

Research conducted in this thesis mainly focuses on analyzing binary executables from x86 platforms. For the next step, it is interesting to see whether we can apply the same methodology towards firmwares from embedded and IoT platforms.

Given the heterogeneity of embedded devices which can exhibit many differences in their microarchitectural details, static disassembling of firmwares is generally considered as more challenging. Comparing to binary executables on x86 platforms, firmwares which may include a kernel and file systems could become much larger and more entangled. In addition, typically one firmware program need to be compiled into executables of different formats regarding various embedded device specifications. Hence, ignorant disassemblers would likely be trapped by even decoding errors.

Most existing security research towards embedded and IoT systems are performed via dynamic analysis; firmwares are executed within a hardware simulator to detect its malicious or vulnerable behaviors. However, dynamic analysis in principle cannot cover every program component. It is also well-known that malicious behaviors could be hidden upon awareness of runtime monitors.

We envision the research opportunity lies within the reliable static disassembling of firmwares. It is interesting to see how we can apply techniques developed within Uroboros to conduct firmware disassembling, and even one step further, deliver *reassembleable disassembling* towards firmwares. A promising application for such static disassembling is to support side channel analysis, where this line of research is considered as significantly more difficult due to the heterogeneity of ARM devices [118]. Additionally, while existing research has conducted study to detect so called "hidden-sensitive operations" from Android apps where malicious activities are protected by control flow guards to evade runtime monitoring [119], to our

best knowledge, firmwares are not analyzable following the same principle. By expanding UROBOROS towards firmware disassembling, our research shall provide a solid foundation to capture malicious firmwares via static analysis.

7.5 Bridge with the LLVM Compiler Framework

We plan to build and maintain a sustainable ecosystem for binary code analysis, and we are particularly interested in linking to the existing ecosystems such as the LLVM compiler framework [120].

We have implemented a number of security applications where commonly-used program analysis techniques (taint analysis, symbolic execution, etc.) are required. Our current practice is to re-build all the involved techniques based on UROBOROS. Things could become even more complex, considering many techniques (e.g., pointer analysis) need to be composed together to construct more rigorous static analysis frameworks such as abstract interpretation [121]. Implementation defect of one single component could lead to the mal-functionality of the whole framework, which could unavoidably counteract the use of UROBOROS in practice.

For the next step, we seek to lift the intermediate representation of UROBOROS into the IR designed by the LLVM community. In some sense, binary code analysis will be directly bridged with the LLVM compiler framework where a rich set of analysis algorithms and utilities are furnished already. In general, both research and engineering efforts need to be committed to lifting our customized low-level representation into the more abstract LLVM IR. Typical research challenges could include the recovery of local variables, abstract stack frames as well as function prototypes. In addition, while our study in Section 2 has shed light on the surprising finding that no tool (including existing work which lifts binary code into LLVM IR [24]) can deliver *reassembleable disassembling*, we follow the same design principle to make the LLVM IR outcome "re-compilable".

Chapter 8
Conclusion

We have presented UROBOROS, a tool that can disassemble stripped binaries and produce *reassembleable* assembly code in a fully automated manner. The key technique implemented in UROBOROS is named *reassembleable disassembling*, in which assembly program is recovered into a *relocatable* format. Moreover, we have extended UROBOROS into a general purpose binary reverse engineering and retrofitting platform that delivers *complete*, *easy-to-use*, *transparent*, and *efficient* instrumentation. UROBOROS provides a rich API and utilities to support analysis and transformations on program internal representations and control flow structures. Our experiments show that reassembled programs by UROBOROS can preserve the original functionality and only incur negligible execution overhead. We also evaluate UROBOROS by comparing it with the state-of-the-art static instrumentation tool regarding several commonly-used instrumentation tasks. Evaluation results show that UROBOROS outperforms the existing tools in terms of lower cost instrumentation and more flexible applications.

We also illustrate the versatility of the UROBOROS instrumentation facilities by developing multiple binary retrofitting and analysis applications for software protection and high-level program representation recovery. We first present FID, a semantics-based function recognition tool in binary code. FID leverages symbolic execution to extract assignment formulas and memory access behaviors. The acquired information will then be used to train a function recognition model with well-performing data mining techniques. Our evaluation shows that FID is comparable with the state-of-the-art tools on normal binaries, and outperforms them on obfuscated binary code.

Software diversification produces different variants of a program, which can effectively defeat code reuse attack and patch-based exploit generation. We initiate a new focus on this area, i.e., *composite software diversification*. Our in-depth study shows that composite diversification can outperform single-pass diversification in terms of better performance. We believe our study can provide useful guidelines for practitioners to design diversification tools in the future.

Appendix | Publication List

[1] Pei Wang, Qinkun Bao, Li Wang, **Shuai Wang**, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation, under review for the 40th International Conference on Software Engineering (**ICSE**), 2018.

[2] **Shuai Wang**, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Analyzing Cache Differences in Production Software for Cache-Based Timing Channels. In *Proceedings of the 26h USENIX Security Symposium* (**USENIX Security**), 2017.

[3] **Shuai Wang** and Dinghao Wu. In-Memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (**ASE**), 2017.

[4] **Shuai Wang**, Pei Wang, and Dinghao Wu. Composite Software Diversification. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution* (**ICSME**), 2017.

[5] **Shuai Wang**, Pei Wang, and Dinghao Wu. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution* (**ICSME**), 2017.

[6] Menghao Li, Wei Wang, Pei Wang, **Shuai Wang**, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. LibD: Scalable and Precise Third-party Library Detection

in Android Markets. In *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering* (**ICSE**), 2017.

[7] **Shuai Wang**, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. Binary Code Retrofitting and Hardening Using SGX. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (**FEAST**), 2017.

[8] Yan Wang, **Shuai Wang**, Pei Wang, and Dinghao Wu. Turing Obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks* (**SecureComm**), 2017.

[9] Pengwei Lan, Pei Wang, **Shuai Wang**, and Dinghao Wu. Lambda Obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks* (**SecureComm**), 2017.

[10] **Shuai Wang**, Pei Wang, and Dinghao Wu. Uroboros: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (**SANER**), 2016.

[11] Pei Wang, **Shuai Wang**, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual Obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy* (**IEEE EuroS&P**), 2016.

[12] Le Guan, Jun Xu, **Shuai Wang**, Xinyu Xing, Lin Lin, Heqing Huang, Peng Liu, and Wenkee Lee. From Physical to Cyber: Escalating Protection for Personalized Auto Insurance. In *Proceedings of the 14th ACM Conference on Embedded Networked Sensor Systems* (**SenSys**), 2016.

[13] **Shuai Wang**, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium* (**USENIX Security**), 2015.

Bibliography

[1] McCamant, S. and G. Morrisett (2006) "Evaluating SFI for a CISC Architecture," in *Proceedings of the 15th Conference on USENIX Security Symposium*, USENIX Association, pp. 209–224.

[2] Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993) "Efficient Software-based Fault Isolation," *SIGOPS Oper. Syst. Rev.*, **27**(5), pp. 203–216.

[3] Adl-Tabatabai, A.-R., G. Langdale, S. Lucco, and R. Wahbe (1996) "Efficient and Language-independent Mobile Programs," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, ACM, pp. 127–136.

[4] Graham, S. L., S. Lucco, and R. Wahbe (1995) "Adaptable Binary Programs," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON '95, USENIX Association, pp. 315–325.

[5] Ford, B. and R. Cox (2008) "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC '08, USENIX Association, pp. 293–306.

[6] Abadi, M., M. Budiu, U. Erlingsson, and J. Ligatti (2005) "Control-flow Integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*, CCS '05, ACM, pp. 340–353.

[7] Ansel, J., P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee (2011) "Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 355–366.

[8] Sehr, D., R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen (2010) "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *Proceedings of the 19th USENIX Conference on Security*, USENIX Association, pp. 1–11.

[9] Erlingsson, U., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006) "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 75–88.

[10] Li, J., Z. Wang, X. Jiang, M. Grace, and S. Bahram (2010) "Defeating Return-oriented Rootkits with "Return-Less" Kernels," in *Proceedings of the 5th European Conference on Computer Systems*, ACM, pp. 195–208.

[11] Niu, B. and G. Tan (2014) "Modular Control-flow Integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 577–587.

[12] Zhang, M. and R. Sekar (2013) "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Security Symposium*, USENIX Association, pp. 337–352.

[13] Zhang, C., T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou (2013) "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, IEEE S&P '13, IEEE Computer Society, pp. 559–573.

[14] Ming, J., D. Wu, G. Xiao, J. Wang, and P. Liu (2015) "TaintPipe: Pipelined Symbolic Taint Analysis," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Association, pp. 65–80.

[15] Shacham, H. (2007) "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications Security*, CCS '07, ACM, pp. 552–561.

[16] Buchanan, E., R. Roemer, H. Shacham, and S. Savage (2008) "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, ACM, pp. 27–38.

[17] Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012) "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pp. 21–35.

[18] Hiser, J., A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson (2012) "ILR: Where'd My Gadgets Go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 571–585.

[19] Pappas, V., M. Polychronakis, and A. D. Keromytis (2012) "Smashing the Gadgets: Hindering Return-oriented Programming Using In-place Code Randomization," in *Proceedings of 2012 IEEE Symposium on Security and Privacy*, IEEE S&P, IEEE, pp. 601–615.

[20] "The IDA Pro disassembler," `https://www.hex-rays.com/products/ida/index.shtml`.

[21] Schwartz, E. J., J. Lee, M. Woo, and D. Brumley (2013) "Native x86 Decompilation using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring," in *Proceedings of the 22nd USENIX Security Symposium*, USENIX Association, pp. 353–368.

[22] "Dagger," `http://dagger.repzret.org/`.

[23] "MC-Semantics," `https://github.com/trailofbits/mcsema`.

[24] Anand, K., M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua (2013) "A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 295–308.

[25] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena (2008) "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 10th International Conference on Information and Communications Security*, pp. 1–25.

[26] Brumley, D., I. Jager, T. Avgerinos, and E. J. Schwartz (2011) "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, pp. 463–469.

[27] Muth, R., S. K. Debray, S. Watterson, and K. De Bosschere (2001) "alto: A Link-time Optimizer for the Compaq Alpha," *Softw. Pract. Exper.*, **31**(1), pp. 67–101.

[28] Edwards, A., H. Vo, A. Srivastava, and A. Srivastava (2001) *Vulcan: Binary Transformation In A Distributed Environment*, *Tech. rep.*

[29] De Sutter, B., B. De Bus, and K. De Bosschere (2005) "Link-time Binary Rewriting Techniques for Program Compaction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **27**(5), pp. 882–945.

[30] Caballero, J., N. M. Johnson, S. McCamant, and D. Song (2010) "Binary Code Extraction and Interface Identification for Security Applications," in *Proceedings of the 2010 Network and Distributed System Security Symposium (NDSS '10)*.

[31] Zeng, J., Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu (2013) "Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming," in *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pp. 487–498.

[32] Kolbitsch, C., T. Holz, C. Kruegel, and E. Kirda (2010) "Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 29–44.

[33] "Hex-Rays Decompiler: Manual," `https://www.hex-rays.com/products/decompiler/manual/failures.shtml`.

[34] "LLVM 3.1 Release Notes," `http://llvm.org/releases/3.1/docs/ReleaseNotes.html`.

[35] Lee, J., T. Avgerinos, and D. Brumley (2011) "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proceedings of the 2010 Network and Distributed System Security Symposium*, NDSS '11.

[36] Saxena, P., R. Sekar, and V. Puranik (2008) "Efficient Fine-grained Binary Instrumentationwith Applications to Taint-tracking," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, CGO '08, pp. 74–83.

[37] Zhang, M., R. Qiao, N. Hasabnis, and R. Sekar (2014) "A Platform for Secure Static Binary Instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pp. 129–140.

[38] Laurenzano, M. A., M. M. Tikir, L. Carrington, and A. Snavely (2010) "PEBIL: Efficient Static Binary Instrumentation for Linux," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '10, pp. 175–183.

[39] O'Sullivan, P., K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis (2011) "Retrofitting Security in COTS Software with Binary Rewriting," in *Future Challenges in Security and Privacy for Academia and Industry*, pp. 154–172.

[40] Wang, S., P. Wang, and D. Wu (2015) "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Association, pp. 627–642.

[41] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 190–200.

[42] Bruening, D. L. (2004) *Efficient, transparent, and comprehensive runtime code manipulation*, Ph.D. thesis, Massachusetts Institute of Technology.

[43] Nethercote, N. and J. Seward (2007) "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pp. 89–100.

[44] Zhao, Q., I. Cutcutache, and W.-F. Wong (2008) "PiPA: Pipelined Profiling and Analysis on Multi-core Systems," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, CGO '08, pp. 185–194.

[45] Kemerlis, V. P., G. Portokalidis, K. Jee, and A. D. Keromytis (2012) "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '12, pp. 121–132.

[46] Buck, B. and J. K. Hollingsworth (2000) "An API for Runtime Code Patching," *Int. J. of High Performance Computing Applications*, **14**(4), pp. 317–329.

[47] Harris, L. C. and B. P. Miller (2005) "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, **33**(5), pp. 63–68.

[48] Cohen, F. B. (1993) "Operating System Protection Through Program Evolution," *Comput. Secur.*, **12**(6), pp. 565–584.

[49] Forrest, S., A. Somayaji, and D. Ackley (1997) "Building Diverse Computer Systems," in *HotOS'97*.

[50] Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014) "SoK: Automated Software Diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, IEEE S&P '14, pp. 276–291.

[51] Balakrishnan, A. and C. Schulze (2005) "Code Obfuscation Literature Survey," *CS701 Construction of Compilers*, **19**.

[52] Konstantinou, E. and S. Wolthusen (2008) "Metamorphic Virus: Analysis and Detection," *Royal Holloway University of London*, **15**.

[53] Giuffrida, C., A. Kuijsten, and A. S. Tanenbaum (2012) "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization." in *USENIX Security Symposium*, pp. 475–490.

[54] Team, P. (2003), "PaX address space layout randomization (ASLR)," .

[55] Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh (2004) "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security*, ACM, pp. 298–307.

[56] Kc, G. S., A. D. Keromytis, and V. Prevelakis (2003) "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and Communications Security*, ACM, pp. 272–280.

[57] Technologies, O. (2003), "Code Virtualizer," `http://goo.gl/JLreyQ`.

[58] VMPSofe (2004), "VMProtect," `http://goo.gl/vXlYgy`.

[59] Sharif, M., A. Lanzi, J. Giffin, and W. Lee (2009) "Automatic Reverse Engineering of Malware Emulators," in *Proceedings of 30th IEEE Symposium on IEEE Security and Privacy*, IEEE, pp. 94–109.

[60] Coogan, K., G. Lu, and S. Debray (2011) "Deobfuscation of Virtualization-obfuscated Software: a Semantics-based Approach," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, CCS '11, ACM, pp. 275–284.

[61] Coppens, B., B. De Sutter, and J. Maebe (2013) "Feedback-driven Binary Code Diversification," *ACM Trans. Archit. Code Optim.*, **9**(4), pp. 24:1–24:26.

[62] Collberg, C., S. Martin, J. Myers, and J. Nagra (2012) "Distributed Application Tamper Detection via Continuous Software Updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 319–328.

[63] Kruegel, C., W. Robertson, F. Valeur, and G. Vigna (2004) "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium*, USENIX Security'04, pp. 18–18.

[64] Theiling, H. (2000) "Extracting safe and precise control flow from binaries," in *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pp. 23–30.

[65] Rosenblum, N., X. Zhu, B. Miller, and K. Hunt (2008) "Learning to Analyze Binary Computer Code," in *Proceedings of the 23rd National Conference on Artificial Intelligence*, AAAI '08, AAAI Press, pp. 798–804.

[66] "IDA-FLIRT," `https://www.hex-rays.com/products/ida/tech/flirt.shtml`.

[67] "DynInst-unstrip," `http://www.paradyn.org/html/tools/unstrip`.

[68] Bao, T., J. Burket, M. Woo, R. Turner, and D. Brumley (2014) "ByteWeight: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pp. 845–860.

[69] Shin, E. C. R., D. Song, and R. Moazzezi (2015) "Recognizing Functions in Binaries with Neural Networks," in *24th USENIX Security Symposium*, USENIX Association, pp. 611–626.

[70] Williams, W. R., X. Meng, B. Welton, and B. P. Miller (2015) "Dyninst and MRNet: Foundational Infrastructure for Parallel Tools," in *Proceedings of the 9th Annual Parallel Tools Workshop*.

[71] Horspool, R. N. and N. Marovac (1980) "An Approach to the Problem of Detranslation of Computer Programs," *Comput. J.*, **23**(3), pp. 223–229. URL `http://dx.doi.org/10.1093/comjnl/23.3.223`

[72] Balakrishnan, G. (2007) *WYSINWYX: What You See is Not What You eXecute*, Ph.D. thesis, University of Wisconsin-Madison.

[73] "Introduction to x64 Assembly," `https://software.intel.com/en-us/articles/introduction-to-x64-assembly/`.

[74] Salwan, J. (2012) "ROPgadget tool," *`http://shell-storm.org/project/ROPgadget`*.

[75] Wang, S., P. Wang, and D. Wu (2016) "UROBOROS: Instrumenting Stripped Binaries with Static Reassembling," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pp. 236–247.

[76] Nanda, S., W. Li, L.-C. Lam, and T.-C. Chiueh (2006) "BIRD: binary interpretation using runtime disassembly," in *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, CGO '06, pp. 358–370.

[77] Deng, Z., X. Zhang, and D. Xu (2013) "BISTRO: Binary Component Extraction and Embedding for Software Security Applications," in *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS '13)*.

[78] Wang, S., P. Wang, and D. Wu (2017) "Semantics-Aware Machine Learning for Function Recognition in Binary Code," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME '17, pp. 388–398.

[79] Van Emmerik, M. and T. Waddington (2004) "Using a Decompiler for Real-World Source Recovery," in *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, IEEE Computer Society, pp. 27–36.

[80] Guilfanov, I. (2008) "Decompilers and beyond," *Black Hat USA*.

[81] Khoo, W. M., A. Mycroft, and R. Anderson (2013) "Rendezvous: A Search Engine for Binary Code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, IEEE Press, pp. 329–338.

[82] Erlingsson, U., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006) "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 75–88.

[83] Davi, L., A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi (2012) "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." in *Proceedings of the 2012 Network and Distributed System Security Symposium*.

[84] Egele, M., M. Woo, P. Chapman, and D. Brumley (2014) "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Association, pp. 303–317.

[85] Pewny, J., B. Garmany, R. Gawlik, C. Rossow, and T. Holz (2015) "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 709–724.

[86] "BinDiff," `http://www.zynamics.com/bindiff.html`.

[87] Hex-Rays, S. (2014), "IDA Pro: a cross-platform multi-processor disassembler and debugger," .

[88] Junod, P., J. Rinaldini, J. Wehrli, and J. Michielin (2015) "Obfuscator-LLVM – Software Protection for the Masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, IEEE, pp. 3–9.

[89] "ByteWeight with Machine Byte Code Features," `https://github.com/BinaryAnalysisPlatform/bap`.

[90] "ByteWeight with Assembly Instruction Features," `http://security.ece.cmu.edu/byteweight/`.

[91] De Moura, L. and N. Bjørner (2008) "Z3: An Efficient SMT Solver," in *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08, pp. 337–340.

[92] Gao, D., M. K. Reiter, and D. Song (2008) "BinHunt: Automatically Finding Semantic Differences in Binary Programs," in *Proceedings of the 4th International Conference on Information Systems Security*, pp. 1–12.

[93] Ming, J., M. Pan, and D. Gao (2013) "iBinHunt: Binary Hunting with Inter-procedural Control Flow," in *Proceedings of the 15th International Conference on Information Security and Cryptology*, pp. 92–109.

[94] Caliskan-Islam, A., R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt (2015) "De-anonymizing Programmers via Code Stylometry," in *24th USENIX Security Symposium*, USENIX Association, pp. 255–270.

[95] Vishwanathan, S. V. N. and A. J. Smola (2002) "Fast Kernels for String and Tree Matching," in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, pp. 585–592.

[96] Zhang, K. and D. Shasha (1989) "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," *SIAM J. Comput.*, **18**(6), pp. 1245–1262.

[97] Balakrishnan, G. and T. Reps (2004) "Analyzing memory accesses in x86 executables," in *Compiler Construction*, Springer, pp. 5–23.

[98] "IDA-Pro 6.7," `www.hex-rays.com/products/ida/6.7/index.shtml`.

[99] "IDA-Pro 6.8," `www.hex-rays.com/products/ida/6.8/index.shtml`.

[100] "IDA-Pro 6.7," `www.hex-rays.com/products/ida/6.7/index.shtml`.

[101] Wang, S., P. Wang, and D. Wu (2017) "Composite Software Diversification," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME '17, pp. 284–294.

[102] Designer, S. (1997), "Getting around non-executable stack (and fix)," .

[103] Wojtczuk, R. (2001) "The advanced return-into-lib (c) exploits: PaX case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*.

[104] Brumley, D., P. Poosankam, D. Song, and J. Zheng (2008) "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, IEEE S&P '08, pp. 143–157.

[105] Linn, C. and S. Debray (2003) "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and Communications Security*, ACM, pp. 290–299.

[106] Triantafyllis, S., M. J. Bridges, E. Raman, G. Ottoni, and D. I. August (2006) "A Framework for Unrestricted Whole-program Optimization," in *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pp. 61–71.

[107] Steyerberg, E. W., M. J. Eijkemans, and J. F. Habbema (1999) "Stepwise Selection in Small Data Sets: A Simulation Study of Bias in Logistic Regression Analysis," *Journal of Clinical Epidemiology*, **52**(10), pp. 935–942.

[108] Wagner, J. M. and D. G. Shimshak (2007) "Stepwise selection of variables in data envelopment analysis: Procedures and managerial perspectives," *European Journal of Operational Research*, **180**(1), pp. 57–67.

[109] Kil, C., J. Jim, C. Bookholt, J. Xu, and P. Ning (2006) "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 339–348.

[110] Davi, L. V., A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi (2013) "Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, ACM, pp. 299–310.

[111] Snow, K. Z., F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi (2013) "Just-In-Time Code Reuse: On the Effectiveness of

Fine-Grained Address Space Layout Randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, pp. 574–588.

[112] Maisuradze, G., M. Backes, and C. Rossow (2016) "What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses," in *Proceedings of the 22nd USENIX Security Symposium*, pp. 139–156.

[113] Dullien, T. and R. Rolles (2005) "Graph-based comparison of executable objects," in *Symposium sur la securite des Technologies de línformation et des Communications*, SSTIC '05.

[114] Flake, H. (2005) "Structural comparison of executable objects," in *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA '05, pp. 161–173.

[115] Bonfante, G., J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry (2015) "CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CSS '15.

[116] Silverstein, J. (1993) "DWARF Debugging Information Format," *Proposed Standard, UNIX International Programming Languages Special Interest Group*.

[117] Oakley, J. and S. Bratus (2011) "Exploiting the Hard-working DWARF: Trojan and Exploit Techniques with No Native Executable Code," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, USENIX Association, pp. 11–11.

[118] Green, M., L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth (2017) "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1075–1091.

[119] Pan, X., X. Wang, Y. Duan, X. Wang, and H. Yin (2017) "Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[120] Lattner, C. (2005) *Macroscopic Data Structure Analysis and Optimization*, Ph.D. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, *See* `http://llvm.cs.uiuc.edu`.

[121] Cousot, P. and R. Cousot (1977) "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252.

Vita

Shuai Wang

Shuai Wang is currently a Ph.D. candidate in College of Information Sciences and Technology, Pennsylvania State University. He is a member of the Software System Security Research Lab where he works with Dr. Dinghao Wu. The overall goal of his research is to enable building more secure software systems. He aims to develop automatic techniques to identify critical vulnerabilities in software systems, and also proposes new methods to protect programs from advanced threats. He received his B.S. degree in Electronic and Information Science and Technology from Peking University in 2012.