BINARY CODE REUSE:

A DYNAMIC ANALYSIS BASED APPROACH

by

Junyuan Zeng

APPROVED BY SUPERVISORY COMMITTEE:

Zhiqiang Lin, Chair

Alvaro Cárdenas

Kevin W. Hamlen

Latifur Khan

To my family.

BINARY CODE REUSE:
A DYNAMIC ANALYSIS BASED APPROACH

by

JUNYUAN ZENG, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2015

ACKNOWLEDGMENTS

I would never have been able to complete my dissertation without the supervision of my advisor and committee members, help and support from friends and my whole family.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Zhiqiang Lin, for his extraordinary guidance, caring and encouragement throughout my entire PhD research. In particular, he guided me into the fantastic world of systems and software security, ignited my passion for security research, and improved the skills of my writing and presentation.

Meanwhile, I am extremely grateful to my committee members, Professor Kevin Hamlen, Professor Latifur Khan and Professor Alvaro Cardenas. This dissertation highly benefits from their insightful reviews and high standards, all of which are absolutely invaluable.

Also, I would like to express my gratitude to all of my friends in S3Lab for their assistance and cooperation. A special thanks goes to Yangchun Fu who always shares with me his creative ideas, system expertise and life experiences. I cherish the time we worked together for each paper deadline. I also thank Erick Bauman for his careful proof reading and feedback for my dissertation.

Finally, this dissertation is dedicated to my grandparents, my parents (Mr. Haixiang Zeng and Mrs. Liudi Liu), and my gorgeous girlfriend (Yingying Zhou), for their support, encouragement, and their best wishes. Without them, I would not be able to pursue my academic goal and finish my PhD dissertation.

May 2015

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

BINARY CODE REUSE:

A DYNAMIC ANALYSIS BASED APPROACH

Publication No. _____

Junyuan Zeng, PhD
The University of Texas at Dallas, 2015

Supervising Professor: Zhiqiang Lin

Binary code reuse aims to extract certain pieces of code from application binaries and make it possible to recompile and relink them with other components to produce new software. With the wide existence of binary code, it is useful to reuse the binary code for different security applications, such as malware analysis and virtual machine introspection. For instance, a malware analyst could reuse proprietary compression and encryption algorithms used in certain malware for message encoding in order to decode their network messages for malware analysis.

In this dissertation, we present a systematic dynamic binary analysis based approach for binary code reuse. In particular, to overcome the challenges for static binary analysis, like obfuscation, this dissertation focuses on applying automated dynamic binary analysis to advance the state-of-the-art of binary code reuse techniques in different aspects. Specifically, a novel solution is presented to generate reusable source code from binary execution traces, featuring obfuscation resilience, free point-to/alias analysis and so on. Meanwhile, in order to facilitate function-level code reuse, this dissertation also proposes a new technique

to automatically recover function interfaces, which can instruct end users to generate and pass appropriate inputs. Finally, since the dynamic execution of our target programs may compromise our analysis, a new dynamic binary instrumentation framework is introduced for the purpose of secure analysis. Compared with the existing platforms, it holds the following advantages: it can perform out-of-VM instrumentation and introspection, it is PIN-API compatible, and it is platform independent.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v
PREFACE vi
ABSTRACT vii
LIST OF FIGURES xiii
LIST OF TABLES xiv
CHAPTER 1 INTRODUCTION 1
1.1 Dissertation Statement 1
1.2 Why Binary Code Reuse is Important 2
1.3 Why Binary Code Reuse is Challenging 4
1.4 Why Dynamic Binary Analysis 5
1.5 Contributions 6
1.6 Scope of This Dissertation 7
1.7 Dissertation Overview 8
CHAPTER 2 DYNAMIC BINARY CODE REUSE: BACKGROUND AND OVERVIEW 11
2.1 The Need of Dynamic Decompilation 11
2.2 The Need of Interface Recovery 12
2.3 The Need of New Instrumentation Platform 14
CHAPTER 3 TOP: CONVERTING EXECUTION TRACES TO REUSABLE SOURCE CODE 17
3.1 Technical Overview 17
3.1.1 Goals and Properties 18
3.1.2 Challenges 19
3.1.3 Architecture Overview 21
3.2 Detailed Design 22
3.2.1 Control Structure Identification 22

3.2.2 Memory Address Symbolization 24
3.2.3 Safety Instrumentation 30
3.2.4 Instruction-to-C Translation 31
3.2.5 Offline Combination 33
3.3 Evaluation 33
3.3.1 Evaluation with Legacy Linux Binaries 34
3.3.2 Evaluation with Obfuscated Windows Binaries 37
3.4 Summary 40
CHAPTER 4 REFIT: RECOVERING OF FUNCTION INTERFACE FOR REUSE 43
4.1 Technical Overview 43
4.1.1 Challenges 43
4.1.2 Problem Statement 46
4.1.3 Architecture Overview 47
4.2 Detailed Design 48
4.2.1 Parameter Discovery 49
4.2.2 Online Type Recovery 51
4.2.3 Dynamic Shape Analysis 56
4.3 Evaluation 59
4.3.1 Overall Results 59
4.3.2 Evaluation with Binary Code Reuse 61
4.4 Summary 63
CHAPTER 5 PEMU: PROVIDING SECURE INSTRUMENTATION 65
5.1 Technical Overview 65
5.1.1 In-VM vs. Out-of-VM Instrumentation 66
5.1.2 Objectives 68
5.1.3 An Example 69
5.1.4 Architecture Overview 70
5.2 Instrumentation Engine 71
5.2.1 TRACE Constructor 74

5.2.2 Code Injector . . . 78
5.2.3 Putting it all together . . . 78
5.3 Introspection Engine . . . 79
5.3.1 Identification of Monitored Process/Threads . . . 79
5.3.2 Addressing the Semantic Gap Challenge . . . 80
5.4 Evaluation . . . 83
5.4.1 Compatibility Testing With Pin Plugins . . . 84
5.4.2 Performance Evaluation . . . 85
5.4.3 Memory Cost Evaluation . . . 87
5.5 Summary . . . 88
CHAPTER 6 APPLICATION . . . 90
6.1 Malware Unpacking and Identification . . . 90
6.2 Security Function Transplanting . . . 92
6.3 Interface recovery of user-defined functions . . . 93
6.4 Shape recovery for recursive data structures . . . 95
6.5 Syscall Tracing for Anti-Analysis software . . . 96
CHAPTER 7 LIMITATIONS AND FUTURE WORK . . . 98
7.1 TOP . . . 98
7.2 REFIT . . . 100
7.3 PEMU . . . 100
CHAPTER 8 RELATED WORK . . . 102
8.1 Decompilation . . . 102
8.2 Dynamic Data Dependency Tracking . . . 102
8.3 Malware Analysis and Unpacking . . . 103
8.4 Binary Code Extraction and Reuse . . . 103
8.5 Data structure reverse engineering . . . 103
8.6 Shape analysis . . . 104
8.7 Memory Forensics . . . 104
8.8 Other DBI Frameworks . . . 105

CHAPTER 9 CONCLUSION 110
REFERENCES 113
VITA

LIST OF FIGURES

1.1 A framework for dynamic binary code reuse from binary 7

3.1 Partial code recovery issue faced by TOP. Assume the provided input is `year_of_service = 15` and `salary = 80,000`. An unsafe code recovered includes lines 1, 2, 3, and 6 of the original program. 20

3.2 Overview of TOP Framework. 21

4.1 Challenges in parameter recognization. 45

4.2 Non-trivial function interface case. 45

4.3 Overview of REFIT. 48

4.4 Examples on argument accessing inside a function. 49

4.5 Locations of Function Parameters. 53

4.6 The Primitive Type Lattice in REFIT. 55

4.7 Different Shapes of the Recursive Data Structures. 56

4.8 Passing Big Structure as Argument. 61

5.1 Differences Between in-VM and Out-of-VM Instrumentation. 67

5.2 A PEMU plugin to count the number of executed instructions. 70

5.3 Architecture Overview of PEMU. 71

5.4 Distributions of PIN APIs. 72

5.5 Detailed Design of Our Instrumentation Engine. 77

5.6 Detailed Steps For An Execution Forwarded Guest Syscall. 83

5.7 Memory Cost Comparison with SPEC2006 Benchmarks 88

6.1 Interface Recovery for `BFEncrypt` 94

6.2 Examples for Shape Recovery 95

6.3 A cross-OS PEMU plugin to trace the syscall. 96

LIST OF TABLES

3.1 Examples of Memory Address Symbolization. 25

3.2 Evaluation results with top-3 binary programs in 7 categories (in terms of recovered source code size). 36

3.3 Evaluation results with 17 obfuscated binary programs. 39

3.4 Disassembling results from IDA Pro and TOP for obfuscated programs. 40

4.1 Summary of the Benchmark Programs 60

4.2 Statistics of the Overall Result of the Benchmark Programs 60

4.3 Statistics of the Recovered Functions (I) 62

4.4 Statistics of the Recovered Functions (II) 62

5.1 Compatibility Testing with Existing PIN Plugins. 84

5.2 Performance compared with vanilla-QEMU and PIN. 86

6.1 Evaluation results with 10 publicly available packers using `tasklist.exe` as a testing goat program. 92

8.1 Comparison with other dynamic binary instrumentation platforms. 106

CHAPTER 1

INTRODUCTION

1.1 Dissertation Statement

In software engineering, source code reuse involves utilizing existing code to build new software without requiring software redevelopment efforts. However, source code sometimes is not available and software is instead only represented in a binary executable format after compilation:

- **Commercial-off-the-shelf Software** is still dominating the industry market. For example, `Windows OS` controls more than `91.02%` of the desktop operating system market share in March 2015 (NETMARKETSHARE, 2015). Besides, a tremendous number of applications are closed source since it is directly related to commercial benefits, software security etc., such as `Microsoft Office` and `PDF Reader`.

- **Legacy applications** are based on languages, platforms, or even methods that are outdated compared to current technologies. Due to certain reasons, such as poor maintenance, some source code is unavailable.

- **Malicious software** is still performing many serious attacks on the Internet, such as `SPAMMING`, `DDOS` and so on. In many cases, in order to prevent malware analysis, the malicious programs are presented in executable format (even with binary obfuscation).

From a reverse engineering perspective, it is natural to ask "*Can we directly reuse the binary code without any source code*?". In other words, we call it Binary Code Reuse. In general, binary code reuse can be categorized into two different areas: *in-context* Binary

Code Reuse and *out-of-context* Binary Code Reuse. In relation to the original binary programs, the former executes the reusable code in the same context while the latter runs the code in a different context (namely, a new context has to be constructed for out-of-context Binary Code Reuse).

In this dissertation, when we use the term Binary Code Reuse, we refer to out-of-context Binary Code Reuse and define it as the process of extracting pieces of code from the original binary programs and recompiling and linking these pieces to generate new software. This facilitates different applications in the areas of software engineering and security.

1.2 Why Binary Code Reuse is Important

As previously mentioned, the source code is not existing in many situations. For example, certain critical legacy applications are required to run continuously on older operating systems, resulting in insecure and outdated configurations and impacting daily operations. In fact, The Defense Advanced Research Projects Agency (DARPA) had launched a project (DARPA, 2011) to solve this problem through innovation in Binary Code Reuse.

In summary, Binary Code Reuse has the following specific impacts on computer security and software development.

- **Function transplanting**– One of the most straightforward applications is to reuse the existing binary code in order to avoid software redevelopment. The main advantage is that it reduces the cost and time of developing new software from scratch in user (Kim et al., 2014) and kernel space (Chipounov and Candea, 2010). In addition, it can effectively eliminate program bugs by utilizing existing bug-free code.

- **Security attacks**– Binary Code Reuse is already used by attackers. For instance, library functions are chained and reused to perform some malicious activities (called

return-into-libc) (Designer, 1997; Nergal, 2001). Based on borrowed code chunks approach and Turing complete functionality, *return-oriented programming* (Shacham, 2007; Buchanan et al., 2008) is designed to reuse a chain of gadgets for malicious purposes. Furthermore, malicious actions can be also conducted by trojans that are camouflaged by code reuse (Lin et al., 2010b).

- **Malware analysis**– Malware (in binary format) may has its own compression and encryption functions for message encryption. The encoded messages are sent via a private channel to the remote client. In this case, if the proprietary decompression and decryption functions can be reused to decompress and decrypt the encoded data, the malware researchers can have a better understanding of the malware behaviors (Caballero et al., 2010; Kolbitsch et al., 2010).

- **Virtual machine introspection**– In virtual machine introspection, in order to extract useful information, detailed knowledge about guest operating systems (such as data structure, system events) is necessary. We refer to it as the semantic gap problem in security research, which is still very challenging. Fortunately, since the context is typically implied in the code, if we can directly (Fu and Lin, 2012; Fu et al., 2014; Saberi et al., 2014) or indirectly (Dolan-Gavitt et al., 2011) reconstruct it by reusing code, the semantic gap problem can be easily solved.

- **Memory forensics**– Existing memory forensics uses signature scanning to uncover data structure instances (e.g., (Lin et al., 2011)), which requires lot of knowledge, such as field offsets, for interpretation. Typically, the code contains the logic for data structure parsing. Once that code can be directly reused in memory, the data structure instances can be easily interpreted (Saltaformaggio et al., 2014).

- **Software retrofitting/patching**– Legacy code and closed source software may contain vulnerabilities. Some vendors may already identify and patch the components with

vulnerabilities. Instead of reinstalling the whole software, the patched components can be reused to replace the vulnerable ones (Deng et al., 2013).

1.3 Why Binary Code Reuse is Challenging

Overall, Binary Code Reuse involves following three main challenges.

- **Source Code Generation** Based on the definition of Binary Code Reuse, we need to extract the code fragments from original programs and generate the source code. In anther word, it's *Source Code Generation*. As we know, the extracted code may access external data and also require the same semantics as the original code. Thus, how to make the extracted code self-contained and retain the same semantics is challenging.

- **Function Interface Recovery** After that, extracted code fragments may not have their function prototypes or function interfaces available, which reveal how to reuse code pieces by passing appropriate inputs. Thus, recovering the prototype for each function component is essential. In this case, we propose *Function Interface Recovery*, aiming to interface the code segments with the parameter list and return value. Unlike high level programming languages (like `C`), the number of parameters is implicit in the binary code. Also, the type information is partially lost after compilation. Thus, how to recover function interfaces is a difficult task.

- **Secure Binary Analysis** Binary instrumentation is usually applied for executable analysis. As we know, binary instrumentation includes dynamic binary instrumentation (DBI) that analyzes the code already existing in memory, and static binary instrumentation (SBI) that performs offline analysis on binary files. The former has security threats; when malware programs are executed, they may compromise DBI. Similarly, for SBI, it suffers from being short of non-bypassability (the added instrumentation

should not be bypassed or subverted by instrumented code) and completeness (instrumentation should be applied to all executed code) which are potential threats for secure SBI (Zhang et al., 2014). Therefore, how to make the instrumentation secure is another big problem.

1.4 Why Dynamic Binary Analysis

In binary program analysis, there are mainly two approaches, *Static* and *Dynamic*.

- **Static binary analysis** analyzes the binary code without executing it. Static analysis platforms (Brumley et al., 2011; Zhang et al., 2014) are designed for model checking, static data-flow analysis, symbolic execution, etc.

- **Dynamic binary analysis** analyzes the binary by executing programs on a real or virtual processor. Many tools (Nethercote and Seward, 2007; Luk et al., 2005; Bungale and Luk, 2007; Yin and Song, 2010; Peter Feiner and Goel, 2012) perform dynamic binary analysis for program profiling, program debugging, dynamic data flow analysis, execution visualisers, etc.

These two approaches are complementary. Static analysis is capable of considering all of the execution paths within one program and thus it is sound and complete. On the other hand, dynamic analysis is not sound since it only considers one single execution path (Collberg et al., 2003). However, because it reveals the real values "in the perfect light of runtime" (Desoli et al., 2002), dynamic analysis is more precise than static analysis and thus make it more straightforward. In particular, with concrete instead of abstract values, dynamic binary analysis is point-to/alias analysis free and even obfuscation resilient (Collberg et al., 1997).

In order to make extracted code executable without crashes and our analysis more simple, we prefer preciseness and straightforwardness rather than completeness. Therefore, instead of static analysis, this dissertation focuses on dynamic analysis.

1.5 Contributions

Our dissertation will address the aforementioned challenges by developing new techniques and platforms for automatic binary code reuse based on dynamic binary analysis, including source code generation, function interface recovery and secure dynamic binary instrumentation. Overall, our contributions can be summarized as follows.

- We present a systematic approach for automatic code reuse from binaries. As illustrated in Figure 1.1, our method includes three key components: REFIT (Automatic Recovery of Function Interface from Binary Code), TOP (Obfuscation Resilient Binary Code Reuse through Trace-oriented Programming) and PEMU (PEMU: A PIN Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework), all of which will provide a complete solution for BINARY CODE REUSE. As a whole, a target executable is the input for our BINARY CODE REUSE framework and the output is the reusable source code. In particular, (1) we execute the target program in our sandbox, PEMU, to ensure safe dynamic instrumentation; (2) atop PEMU, TOP and REFIT run separately, but their results can be combined via *PC* (instruction counter) in an offline phrase to generate the reusable source code.

- TOP is able to extract the code from dynamic traces and generate the reusable source code. In order to make the source code reusable, the corresponding data dependence is also extracted.

- REFIT is responsible for the recovery of function interfaces (parameter list and return value), which, in turn, helps with function-level code reuse and understanding.

- PEMU provides a secure out-of-vm dynamic binary instrumentation (DBI) framework for TOP and REFIT. Besides, it also addresses the challenges of virtual machine introspection and is PIN-API compatible.

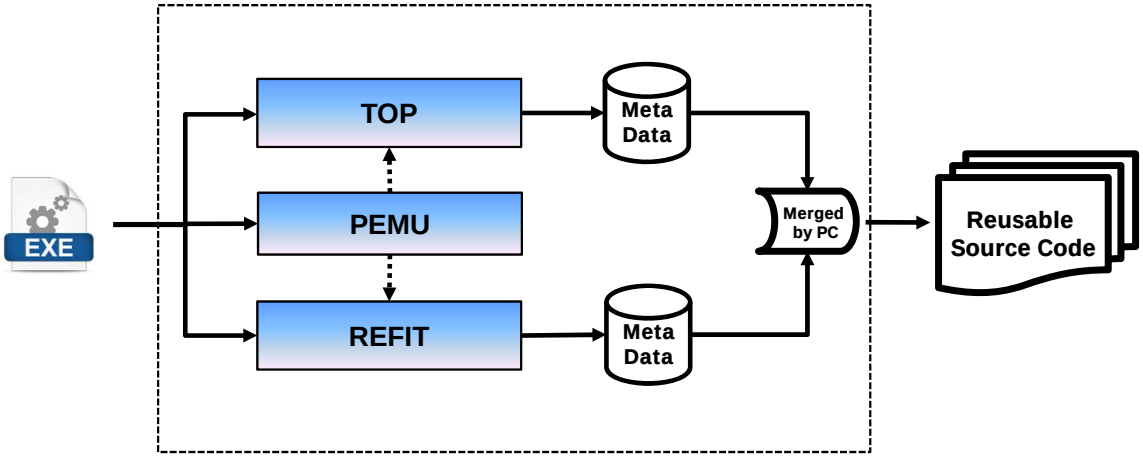

Figure 1.1. A framework for dynamic binary code reuse from binary

- We have implemented our BINARY CODE REUSE framework, and our experimental results show that this framework is highly effective. In particular, TOP is able to successfully generate the reusable `C` source code from over `100` legacy binaries in Linux, `15` representative obfuscated samples from malware in Windows and `10` publicly available packers; REFIT can effectively recover the interfaces for `10` selective programs; PEMU successful evaluates compatibility with over `19` PIN plugins and performance based on the `SPEC2000` benchmark shows about `4.33` time slowdown and `9` M memory overhead.

1.6 Scope of This Dissertation

Our BINARY CODE REUSE system design and implementation has the following assumptions.

- **Architecture**– Our techniques are evaluated on the `x86` platform. Some modifications are required to apply our techniques for other platforms, for instance, the data flow propagation based on instructions needs adjustment.

- **Operating system**– For host operating system, we take Linux (`Ubuntu`) as our operating system while we use either Linux (`Debian`) or Windows as our guest operating system.

- **Programming Languages**– We assume the programs are implemented in `C/C++`. We don't consider the programs implemented by other programming languages (such as Java, Python or Perl) because compared with `C/C++` programs, they have different run time mechanisms.

- **Compilers**– For Linux programs, programs are assumed to be compiled using `gcc`, while in Windows, `TCC` (Bellard, 2004) is our default compiler.

1.7 Dissertation Overview

Our Binary Code Reuse framework contains three components, all of which are targeting the goal of facilitating binary code reuse. More specifically, as shown in Figure 1.1, Pemu provides a safe dynamic binary instrumentation sandbox for both Top and Refit. On top of the secure environment, Refit and Top solve two different sub-problems/challenges of Binary Code Reuse, source code generation and function interface recovery. The output of these components can be merged to output the reusable source code.

An outline of this dissertation is illustrated as follows.

- **Chapter 1** offers an overview of binary code reuse based on dynamic binary analysis, in terms of dissertation statement, motivation, challenges, contribution, etc.

- **Chapter 2** provides a brief introduction for Top, Refit and Pemu, respectively.

- **Chapter 3** first presents our core component, Top, which is able to develop new software from existing binary code. The substantial difference compared with existing work is that Top gains benefits from dynamic analysis (such as being obfuscation resilient, points-to analysis free), and it elevates the low level binary code into high level `C` code. Thus, Top can be used for malware analysis, especially malware function inspection and classification. We have implemented a proof-of-concept Top framework,

and our experimental results with over hundreds of pieces of benign and malicious software have demonstrated that TOP is able to reconstruct source code from execution traces for malware analysis and classification, and function transplanting.

- **Chapter 4** presents the design, implementation and evaluation for our second component, REFIT, which advances the state-of-the-art with several compelling features. In particular, REFIT recovers not only the number of the parameters, but also the non-trivial types such as pointer related types. To facilitate parameter initialization in software development, REFIT reverse-engineers recursive data structures through a dynamic shape analysis. REFIT also features efficiency by making these analyses online, without relying on offline traces.

- **Chapter 5** PEMU is presented in this chapter as our last component, which is a new open source DBI framework that is compatible with PIN-APIs, but supports out-of-VM introspection for both user level processes and OS kernels. Unlike in-VM instrumentation, in which there is no semantic gap, out-of-VM introspection requires bridging the semantic gap and providing abstractions (i.e., APIs) for programmers. One important feature of PEMU is its API compatibility with PIN. As such, many PIN plugins are able to execute atop PEMU without any source code modification. We have implemented PEMU, and our experimental results with the SPEC 2006 benchmarks show that PEMU introduces reasonable overhead.

- **Chapter 6** presents the real-world applications for TOP, REFIT and PEMU.

- **Chapter 7** illustrates the limitations and future work for TOP, REFIT and PEMU separately.

- **Chapter 8** lists the related work for our research.

- **Chapter 9** concludes this dissertation. We end with a discussion of a number of open research problems in this area.

CHAPTER 2

DYNAMIC BINARY CODE REUSE: BACKGROUND AND OVERVIEW

The motivation for our BINARY CODE REUSE is to systematically solve three challenges: reusable source code generation, function interface recovery and secure binary instrumentation.

As shown in Figure 1.1, our solution contains three parts, TOP, REFIT and PEMU. Before illustrations of technical details in Chapter 3, 4 and 5, we firstly provide a brief background and overview for each of them. In particular, we will present the needs for TOP in Section 2.1, then REFIT in Section 2.2 and finally PEMU in Section 2.3.

2.1 The Need of Dynamic Decompilation

Binary code reuse involves extracting selected pieces of code from an application binary, recompiling and linking them with other components, and producing a new software program. Binary code reuse is desirable in many security applications such as malware code inspection and classification, legacy binary program retrofitting, security function transplanting, and source code recovery.

While decompilation (Cifuentes, 1994; Cifuentes and Gough, 1995) has long been adopted for source code recovery and hence binary code reuse, it is based on static analysis and has limitations. For example, a state-of-the-art decompilation tool, Hex-Rays (Hex-Rays, 2005) is able to generate C code that is semantically equivalent to the original binary code in terms of execution effects. It was used to recover the Stuxnet source code in 2011 (Constantin, 2011). However, the source code generated by Hex-Rays may be unsafe. More

specifically, it does not fully recover non-trivial indirect jump targets or function pointers, due to its static analysis nature; nor does it deal with binaries with *obfuscation*, such as instruction aliasing (Aliasing, 2012) or encrypted binary code case (e.g., those obfuscated by packers). While there are recent advances for improving static decompilation, such as semantic-preserving structural analysis (Schwartz et al., 2013), some challenges still remain. Moreover, the recovered source code very often may not be directly compilable.

Recent efforts – notably BCR (Caballero et al., 2010) and Inspector Gadget (Kolbitsch et al., 2010) – can extract part of a malware binary for reuse or inspection. However, they are also not safe as they rely on incomplete dynamic analysis without a reliable mechanism to remedy the partial information acquired from the dynamic execution. Virtuoso (Dolan-Gavitt et al., 2011) is a technique that extracts execution traces and translates them to executable python programs that could run outside the platform. In addition to similar limitations, it requires special runtime support – a python interpreter – to execute and often incurs high performance overhead (up to hundreds of times of slowdown). VMST (Fu and Lin, 2012) addresses Virtuoso's limitations but it only supports kernel functions and requires a heavy-weight dual-VM architecture (Fu and Lin, 2013).

Based on the emerging needs, we designed and implemented a new dynamic decompiler, Top, which is able to generate the reusable source code from dynamic execution traces. Compared with existing approaches, Top has the advantages of obfuscation resilient, point-to/alias analysis free and so on. We will list the technical details in Chapter 3.

2.2 The Need of Interface Recovery

Being the final representation of software, binary code is everywhere. Regardless of system software (e.g., operating system, hypervisor, compilers), application software (e.g., web-browser, PDF-viewer, media-player), or even malicious software (e.g., virtus, trojan, rootkits, bots), eventually they are all represented in the binary form. Since there are plenty of binary

code in a computer system, it would significantly save programmer's efforts if we were able to reuse the legacy binary code when constructing new software (i.e., binary code reuse). While decompiler (Cifuentes, 1994; Emmerik and Waddington, 2004; Breuer and Bowen, 1994) has been earlier used for this purpose, recently there are a number of other techniques motivated from different applications for binary code reuse.

From *malware analysis* perspective, Caballero et al. proposed BCR (Caballero et al., 2010) that makes a first systematic study towards automated binary code reuse and demonstrates its effectiveness by extracting encryption and decryption components from malware code. Similarly, Kolbitsch et al. proposed Inspector Gadget (Kolbitsch et al., 2010) that uses dynamic slicing to extract and reuse certain features inside malware programs.

From *malware development* perspective, binary code reuse was initially used for developing the shell code, such as in return-into-libc (Designer, 1997) and return-oriented-programming (Shacham, 2007) which reuses the instruction sequence building block. Meanwhile, unlike these primitive reusing, our prior work (Lin et al., 2010b) demonstrated that we can also reuse the legal binary code, in particular their calling context, to create stealthy trojans through directly patching the benign software.

From *system administration* perspective, Dolan-Gavitt et al. proposed Virtuoso (Dolan-Gavitt et al., 2011), a technique for better virtual machine introspection (VMI) (Garfinkel and Rosenblum, 2003). The basic idea is to acquire traces of an inspection command (e.g., `ps`) on a clean guest OS through dynamic slicing. Then extracting these sliced code and executing them at the VMM layer to introspect the identical version of the guest OS that may be compromised.

Therefore, binary code reuse has many benefits and has been widely investigated in many applications and reused in different granularities from binary instruction sequences level (i.e., gadget level (Shacham, 2007; Kolbitsch et al., 2010)), or function components level (Dolan-Gavitt et al., 2011; Caballero et al., 2010), or execution context level (Lin et al., 2010b; Fu

and Lin, 2012). While gadget level binary code reusing may not need to precisely recover the function interfaces, component or execution context reusing often needs to infer the function interfaces.

A function interface reveals how a function gets called, how many parameters are needed (note that global variables are implicitly considered as parameters), what the syntactic (i.e., the layout and shape of a data structure) and semantic (i.e., the meanings) type is for each parameter as well as the return value. Recovering of function interfaces is also valuable for reverse engineering of binary code (or program understanding in general), as it shows the interconnections between each function. In addition, it will also enable the specification recovery for APIs when their documentation is lost.

Unfortunately, existing techniques tend to be ad-hoc or require manual efforts when dealing with function interface reverse engineering. They either ignore complex semantics type recovery (e.g., BCR (Caballero et al., 2010) left this as their future work), or use simple heuristics that cannot handle the general cases in syntactic recovery, e.g., REVNIC (Chipounov and Candea, 2010) uses a heuristic that parameters are in the form of a positive offset with `ebp` to identify the parameters, which is not always true.

We, therefore, designed and implemented our approach for function interface recovery, Refit, which can successfully dynamically interface the binary code with function protocols, including syntactic, semantics and pointer shape for return value and parameter list. More technical details will be listed in Chapter 4.

2.3 The Need of New Instrumentation Platform

Dynamic binary instrumentation (DBI) is an extremely powerful technique for program analysis. At a high level, it dynamically inserts extra analysis code into the running binary program to observe how it behaves. It works similarly to a debugger but the analysis routine is programmed. Therefore, it can be used to automatically inspect the program state at

instruction level and build many program analyses, such as performance profiling (e.g., (Wu et al., 2005; Wallace and Hazelwood, 2007)), architecture simulation (e.g., (Narayanasamy et al., 2006)), program debugging (e.g., (Lu et al., 2006)), program shepherding (e.g., (Kiriansky et al., 2002)), program optimization (e.g., (Bala et al., 2000)), dynamic data flow analysis (e.g., taint analysis (Newsome and Song, 2005; Schwartz et al., 2010)), reverse engineering (e.g., (Lin et al., 2010a)), and malware analysis (e.g., (Egele et al., 2007; Yin et al., 2007)).

Today, there are many DBI platforms such as Pin (Luk et al., 2005), Valgrind (Nethercote and Seward, 2007), DynamoRIO (Bala et al., 2000), Qemu (Bellard, 2005a), and Bochs (community, 2001). Each platform is built atop its own virtual machine (VM), and has its own pros and cons. For example, process-level DBI such as Pin and Valgrind provides rich APIs to analyze user level binary code execution, but the analysis code is executed inside the VM (i.e., in-VM) with the same privilege as the instrumented process. Moreover, it does not support any kernel-level code instrumentation. Some platforms only support a limited type of operating system (OS), e.g., Valgrind only supports Linux binaries but provides no support for Microsoft Windows binaries. Some platforms are designed as a full system emulator (e.g., Qemu), but do not provide any general DBI APIs. As such, *can we build a cross-OS, API-rich, out-of-VM dynamic binary instrumentation framework that supports both user level and kernel level code?*

While there have been attempts to address this problem, they only partially achieved these goals. Specifically, PinOS (Bungale and Luk, 2007) attempted to create a kernel instrumentation tool atop the Xen (Barham et al., 2003) hypervisor. However, it only supports inspecting the very low level instruction semantics (such as the executing instruction address), and does not support any high level instrumentation and introspection (e.g., get the running process ID inside the VM). Meanwhile, because of its implementation of stealing memory from the guest OS, it does not offer strong isolation and the analysis routine can

be accessed by the instrumented process or kernel. Another attempt is TEMU (Yin and Song, 2010), which extends QEMU with its own APIs to allow end-users to develop TEMU-plugins for whole system instrumentation. Though it has greatly reduced developers' efforts in understanding the internals of QEMU in order to develop any useful plugins, it has only limited APIs compared to those provided by PIN. A recent effort, DRK (Peter Feiner and Goel, 2012), is able to perform kernel instrumentation. However, it is still an in-VM solution and does not isolate the analysis code (the analysis routine is executed as Linux Kernel modules), resulting in security issues when the kernel has malware.

Thus, we designed and implemented a new open source DBI framework, PEMU, featuring PIN-APIs compatible, secure instrumentation for kernel and user space and out-of-vm introspection. We present this works in detail in Chapter 5.

CHAPTER 3

TOP: CONVERTING EXECUTION TRACES TO REUSABLE SOURCE CODE[1]

In this chapter, we illustrate our reusable source code generation approach, which is implemented as Top (Zeng et al., 2013), in terms of technical overview in Section 3.1, detailed design in Section 3.2, evaluation in Section 3.3 and summary in Section 3.4.

3.1 Technical Overview

We present a new binary code reuse approach called *trace-oriented programming* (Top). The basic idea of Top is that, given a binary (possibly obfuscated), through dynamic analysis of its execution, we collect instruction traces and translate the executed instructions into a high level program representation using C with templates and inlined assembly (for better performance). Such a program representation can be directly compiled and linked with other code to produce new software. Unlike static analysis-based decompilation, Top is based on dynamic analysis. Therefore, it has better resilience against obfuscation (Collberg et al., 1997) and it does not require any binary points-to (Burke et al., 1995; Liang and Harrold, 1999; Pearce et al., 2007) or alias analysis (Debray et al., 1998; Brumley and Newsome, 2006).

A by-product of Top is the ability to instrument the newly generated code with additional guards, thereby gaining the ability to mitigate the incompleteness caused by dynamic

[1]

analysis. In particular, since TOP targets binary code reuse, where the recovered code needs to run independently or be linked with other programs, it must ensure that the newly generated reusing code (i.e., the reusable software component) reproduces the same behavior with the traces and flag exceptions if the behavior is not consistent with the traces. In other words, there is no gray area in the newly generated code, with its behavior well-defined by the traces. By instrumenting the code while reconstructing its source code, TOP is able to achieve such behavior consistencies.

In this section, we first present the goals and properties in Section 3.1.1, then list the challenges in Section 3.1.2 and finally provide an architecture overview in Section 3.1.3.

3.1.1 Goals and Properties

Targeting binary code reuse, TOP differs from the traditional static analysis-based decompilation techniques in that it performs *dynamic decompilation of program traces*. The salient properties of TOP are as follows:

- **Resilient against obfuscation.** For intellectual property protection or anti-malware analysis purpose, many binary programs are obfuscated. Obfuscation techniques (Collberg et al., 1997) range from instruction aliasing, garbage code insertion, register reassignment, instruction substitution, to binary code encryption and packing (Guo et al., 2008), and even virtualization-based obfuscation (Sharif et al., 2009). However, no matter how complicated the obfuscation technique is, the obfuscated program has to be executable. From its execution traces, we can recover the program's source code, although it may not be exactly the same as the original code (e.g., those obfuscated with virtualization), and reuse it in new programs.

- **Free from Point-to and Alias Analysis.** It is challenging to perform point-to or alias analysis (Burke et al., 1995; Liang and Harrold, 1999; Pearce et al., 2007; Debray

et al., 1998; Brumley and Newsome, 2006) statically. However, since TOP is based on dynamic execution, the point-to relations are exercised directly, hence there is no need to perform complicated point-to analysis to figure out pointers' targets.

- **Concrete Instead of Abstract Values.** Similar to concolic testing (Godefroid et al., 2005; Sen et al., 2005) (a mix of symbolic and concrete execution), concrete values are sometimes preferred in some reuse scenarios. By using TOP, values of the variables accessed during the program's execution can be observed. When generating the new source code, some arguments can be replaced with concrete values to avoid certain complicated tasks, such as environment setup and pre-condition computation; and a function pointer can be replaced with a concrete function call if the pointer always points to a specific function in the trace.

3.1.2 Challenges

It is a well-established approach to translate machine code back to human readable disassembled code (which is called disassemble). However, from disassembled code to high-level code (i.e., decompilation (Cifuentes, 1994; Cifuentes and Gough, 1995; Mycroft, 1999)), there is no standard approach and no significant breakthrough in the past few years for native code (although decompilation is more successful for other low-level code such as Java-bytecode (Miecznikowski and Hendren, 2002)).

TOP is built atop dynamic binary instrumentation (DBI) which can be used to generate execution traces. However, the traces from DBI-tools (e.g., PIN (Luk et al., 2005) and QEMU (Bellard, 2005b)) cannot be reused directly, and we must solve the following challenges:

- **Control Structure Identification.** Normally a trace is a sequence of instructions executed by the CPU, and there is no explicit control structure inside. For example,

the loop is unrolled, and callers and callees become sequential in the trace. Therefore, we have to identify the program control structures such as loops and function calls for the C code being generated.

- **Type Classification of Literal Values.** The literal value (i.e., the immediate value) in an instruction can be associated with different types, such as a global or read-only data address, a function pointer or a constant. If the literal value refers to a memory access or a function pointer, it has to be converted to the address which is associated with the generated C code instead of the original binary code. Therefore, we have to develop techniques to precisely differentiate the types of the literal values and symbolize them.

- **Safe Reuse.** Based on dynamic analysis, TOP faces the code coverage challenge. However, our goal is to make the non-exhaustively executed code reusable and ensure that the recovered code is safe and consistent with the traces. For example, as shown in Figure 3.1, if we do not have any safety check in the partially executed code, the original semantics of the program may get violated. Thus, we have to develop techniques to ensure the safety of the recovered source code.

```
1 if (year_of_service > 10) {
2     if (salary < 100000.0)
3         salary = 100000.0;
4     else
5         salary = salary*1.02;
6 }
```

(a) Original Source Code

```
1 if (year_of_service > 10) {
2     if (salary < 100000.0)
3         salary = 100000.0;
6 }
```

(b) Naively Recovered Source Code from Trace

Figure 3.1. Partial code recovery issue faced by TOP. Assume the provided input is `year_of_service = 15` and `salary = 80,000`. An unsafe code recovered includes lines 1, 2, 3, and 6 of the original program.

3.1.3 Architecture Overview

An architecture overview of TOP is presented in Figure 3.2. The input to TOP is the application binary code and a test suite; and the output is the executed, modularized, and reusable components with C source code representation, which can then be directly compiled and linked to generate new programs.

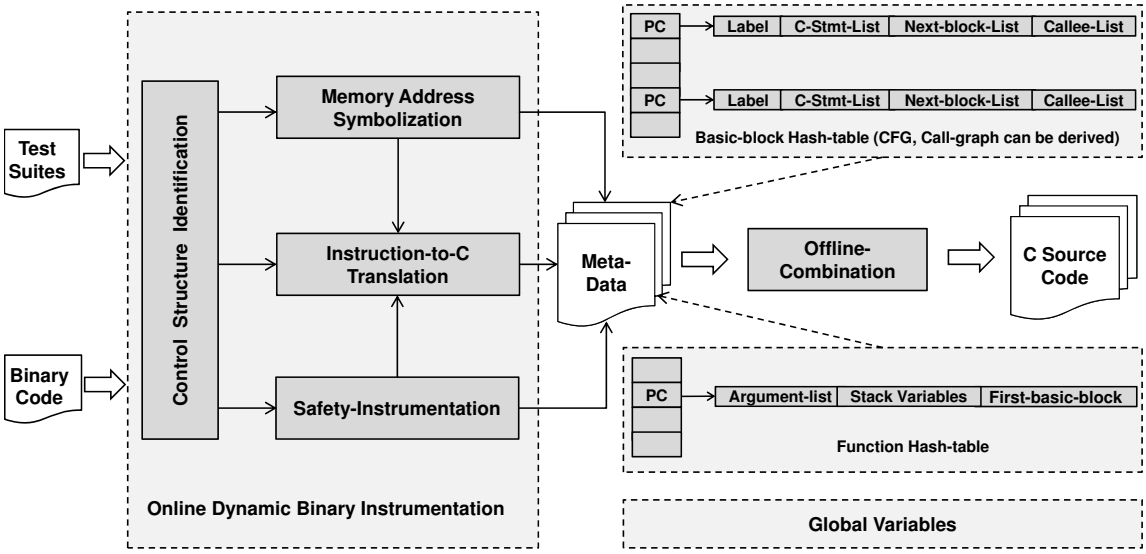

Figure 3.2. Overview of TOP Framework.

There are five key components in TOP, four of which are designed using online DBI. More specifically, through dynamically monitoring executed instructions, we will identify the control structure of the code (Section 3.2.1) such as function calls, loops, and branches; discover the literal value types and relocate them with symbols (if necessary) such that they can be recompiled (Section 3.2.2); collect the control flow graph (CFG) for safety instrumentation (Section 3.2.3); and translate instructions using inline assembly for C code (Section 3.2.4). Finally, we will have all the meta-data necessary to reconstruct the C code. The offline-combination component (Section 3.2.5) will combine multiple runs if necessary and emit the final C code.

3.2 Detailed Design

3.2.1 Control Structure Identification

The typical control structures in binary code include sequence, selection or conditional branch (`jcc`, namely jump if condition is met; there are 77 such instructions in x86 (Intel, 2012)), and repetition (i.e., loop). However, loop is essentially a special case of conditional branch which has a backward edge in the control flow graph (CFG). In other words, at the binary level, we only see instructions being executed either sequentially, or with a control flow transfer (including explicit ones such as `jcc/jmp/ret/call` and implicit ones such as `push/ret`).

Our goal is to translate binary code (more precisely the dynamically disassembled code) into C code that can be recompiled. In fact, control structures are already encoded in the binary, especially in the program's CFG, where each node represents a basic block (BB) in which instructions are executed sequentially; and a directed edge represents the control flow transfer (such as `jcc/jmp`).

CFG is not only crucial to *control structure identification*, but also important for our *safety instrumentation* technique (Section 3.2.3) with the goal of tolerating incomplete execution (i.e., only one of the two branches of a conditional gets executed and dynamically decompiled). Therefore, the first step of TOP is to dynamically build a program's CFG based on executed instructions.

Dynamic CFG Construction. Since we target programs that may be obfuscated or metamorphic, we cannot rely on static analysis (e.g., static disassembler). Hence, our key idea for CFG construction is to leverage the exercised instruction addresses to safely connect the BBs (with their successors and predecessors) during the program's execution. In particular, we focus on handling metamorphism that may cause instructions being dynamically modified or replaced (e.g., those obfuscated by packers). The procedure for dynamic CFG construction is presented in Algorithm 1.

```
1: Input: PC, the current executing instruction address; Shadow: the shadow memory of the accessed program address
   space, which stores the current memory overwriting layers.
2: Output: Partial CFG stored in the basic block hash table (BBHT)
3: DynamicBinaryInstrumentation (PC, BB) {
4:     if (Inst(PC) ∈ {jcc, jmp, call})
5:         next_pc ← Next({jcc, jmp, call})
6:         layer ← GetShadowLayer(Shadow, next_pc)
7:         N ← GetBB(next_pc, layer);
8:         T ← Set of exercised successor of PC;
9:             if (N ∉ T)
10:                Insert N to BB's successor in BBHT;
11:                Insert BB as N's predecessor in BBHT;
12:    else if (Inst(PC) ∈ {ret})
13:        next_pc ← Stack(ESP),
14:        layer ← GetShadowLayer(Shadow, next_pc)
15:        N ← GetBB(next_pc, layer);
16:        T ← Set of exercised successor of PC;
17:            if (N ∉ T)
18:                Insert N to BB's successor in BBHT;
19:                Insert BB as N's predecessor in BBHT;
20:    else if (Inst(PC) ∈ {Memory Write})
21:        Increase the layer by one in the shadow memory.
22:    Append PC to BB if this PC has not been visited;
23: }
```

Algorithm 1: *Dynamic CFG Construction*

For a given executing instruction with address PC, if the instruction is `jcc/jmp/call` (line 4 -11), we will first get the PC of the next instruction (line 5) and retrieve its current layer. The reason to introduce a layer (an unsigned integer) for each PC is to track the version of the newly generated code, and use $\langle PC, layer \rangle$ as the unique index to access a basic block hash table (BBHT). Without layers, we cannot have a one-to-one mapping because the same address can be overwritten and one PC can be mapped to different instructions at different times (e.g., in different unpacking phases if the program is packed by multiple packers). Next, we will retrieve the next BB (i.e., N) of $\langle next\_pc, layer \rangle$ (line 7); if N does not exist, `GetBB` will create one. After that, it retrieves all the successors of current PC, and updates the successor and predecessor of the current BB (line 10-11). Handling of `ret` is similar, except that we have to fetch its next PC from the stack (line 13). Also, for memory write instructions, since we need to track the layer of a memory address, we retrieve the current layer for the overwritten address and increase it by one. Then whenever an instruction is fetched from the memory, we can retrieve its layer from the shadow memory. At the end of the execution of the current instruction, we will append the PC to the current BB if it is

visited for the first time (line 22). Note that our BBHT contains all the control flow transfer information of the executed instructions, all of our following components will leverage the information collected in BBHT as illustrated in Figure 3.2.

3.2.2 Memory Address Symbolization

During program execution, many instruction and memory data addresses become concrete. For instance, the target of a control flow transfer instruction is a concrete instruction address, and the target of a memory access operation is also a concrete address. However, in the recovered code, we often cannot directly use these concrete addresses. Instead, we have to symbolize and relocate them. We call this procedure *Memory Address Symbolization*.

In general, we have to perform (1) instruction operand symbolization and (2) global data symbolization. We do not need to symbolize heap or stack memory addresses because they are dynamically allocated and intrinsically relocatable. For instruction operand symbolization, we need to rewrite a concrete address or literal value with a symbolized address (such as a label). For instance, as shown in the first example in Table 3.1, we need to rewrite the target address of direct `call` with the address of `func_dest_addr`; and the destination address of `je` with label `L_dest_addr`. For global data symbolization (the third and fourth examples in Table 3.1), we need to rewrite the instructions that use concrete addresses of the global variables. Also, we need to rewrite the initial value stored in a global variable if the value itself is an address of a global variable or a function pointer.

Unfortunately, memory address symbolization is challenging. This is because literal values are widely used in many instructions. Many of them are not addresses even though some of them look like so. Only from the later data use of a value, we can infer if it is a memory address, a pointer (with its content being a memory address), or a double pointer (from a two-layer pointer dereference). Also, there could be indirect `call`/`jmp` or even `ret` (as shown in the second example in Table 3.1). We have to back track to decide whether their target

Table 3.1. Examples of Memory Address Symbolization.

Examples		Original Assembly Code	Address Symbolized Code

Instruction Operand Symbolization	Direct Symbolization	`1: je 0x401175`
		`2: call 0x401028`
		`3: mov DWORD PTR 0x424a30, 0x1`
			`1: je L_0x401175`
			`2: call func_0x401028`
			`3: mov DWORD PTR [global_data+0x0], 0x1`

	Indirect Symbolization	`4: push 0x401058`
		`5: ret`
		
		`6: mov eax, 0x409200`
		`7: call eax`
			`4: push OFFSET L_0x401058`
			`5: ret`
			
			`6: mov eax, OFFSET func_0x409200`
			`7: call eax`

Global Data Symbolization	Implicitly Initialized Global Data	`8: mov eax, 0x424a38`
		`9: mov ecx, DWORD PTR [eax]`
		`10: mov edi, ecx`
		`11: mov ebx, DWORD PTR [edi]`
		
		`12: call DWORD PTR 0x424a3c`
			`8: mov eax, OFFSET global_data+0x8`
			`9: *(int*)(global_data+0x8) = global_data+0x4`
			`10: mov ecx, DWORD PTR [eax]`
			`11: mov edi, ecx`
			`12: mov ebx, DWORD PTR[edi]`
			
			`13: *(int*)(global_data+0xc) = func_0x40102d`
			`14: call DWORD PTR [global_data+0xc]`

	Explicitly Initialized Global Data	`13: mov DWORD PTR 0x424a38, 0x424a30`
		`14: mov ecx, DWORD PTR 0x424a38`
		`15: mov DWORD PTR [ecx],0x4`
		
		`16: mov DWORD PTR 0x424a3c, 0x401032`
		`17: call DWORD PTR 0x424a3c`
			`15: mov DWORD PTR [global_data+0x8],OFFSET global_data+0x0`
			`16: mov ecx, DWORD PTR [global_data+0x8]`
			`17: mov DWORD PTR [ecx],0x4`
			
			`18: mov DWORD PTR [global_data+0xc],func_0x401032`
			`19: call DWORD PTR [global_data+0xc]`

operand is directly or indirectly derived from any literal value representing a global address. To this end, we adopt a data dependence tracking (i.e., taint analysis (Chow et al., 2004; Newsome and Song, 2005; Crandall et al., 2006)) algorithm to resolve the instructions whose operands need rewriting, and the global memory addresses that need to be updated with new symbols. In the following, we present the detailed design of our algorithm.

Taint Sources

The goal of our algorithm is to (1) pinpoint the instructions that have a memory operand that needs to be symbolized and (2) pinpoint every global data location that stores an initial function pointer or a global data pointer which need to be symbolized. Therefore, *our taint source contains the instruction address (PC) whenever an instruction involves a literal value operand, or a data value that appears to be a global pointer or a function pointer.* For either case, we are able to determine the operand or the data value is indeed a global/function pointer by observing that it gets dereferenced at a later point. We replace the origin of the operand, which is indexed by the taint source using its PC, with a symbol; or the origin of the memory data indexed by the taint source using its value with a symbol. We will use the examples in Table 3.1 to illustrate how our analysis works.

Consider the third example in Table 3.1 (line 8-11 in the third column). When line 8 gets executed, we will assign the shadow record for `eax` as S[`eax`]=8 (PC) because "`mov eax, 0x424a38`" has a literal value. Instruction address 8 will be used later to update the source operand 0x424a38 at location 8 if the taint sink point (discussed below) indicates that `eax` stores a memory address. When line 9 "`mov ecx, DWORD PTR [eax]`" is executed, it is a taint sink point as it dereferences the value stored in `eax`. We hence know that `eax` stores a global memory address. We then update the operand at address 8, and generate an address-symbolized instruction "`mov eax, OFFSET global_data+0x8`" as shown in line 8 in the last column in Table 3.1. Also, this instruction denotes a new taint source as the value

stored in `eax`=0x424a38 is 0x424a34 that appears to be a pointer pointing to the global area. Hence, the memory address of the source operand is assigned to the shadow record of `eax`, S[`ecx`]=0x424a38.

Taint Propagations

Much like all other taint analysis, the taint record gets propagated along with data movement instructions, and data arithmetic (because of pointer value computation) instructions. For instance, along with data movement instruction at line 10 "`mov edi, ecx`", S[`ecx`]=0x424a38 will be propagated to `edi` (i.e., S[`edi`]=0x424a38).

Taint Sinks

A taint sink is an instruction point that can reveal the type of the operand: whether a literal value is a global memory address or relative address for `call`/`jmp` target; or whether the involved memory operand is a pointer. Therefore, nearly all instructions are possible taint sinks. When a taint sink instruction is executed, we will update the assembly code depending on whether it involves instruction operand symbolization or global data symbolization.

Case I: Instruction Operand Symbolization. There are many instructions that take literal values. For some instructions we can immediately infer that the operand is a memory address (we call it direct symbolization). For the others we must infer based on later data use (we call it indirect symbolization). Therefore, we have the following two strategies:

- **Direct Symbolization.** Direct symbolization applies to the direct control flow-related instructions (i.e., `call`/`jmp`/`jcc`). In particular, if the operand of such an instruction is a literal value, we will directly symbolize it. For instance, as shown in line 1-2 of the first example in Table 3.1, we will directly symbolize `0x401175` with a label `L_0x401175` for the `je` instruction; and `0x401028` with `fun_0x401028` for the `call`

instruction. Direct symbolization is critical for unpacking. Usually the last step when finishing unpacking is through a control flow transfer instruction (`jmp/call/ret`). If we do not symbolize the target address of the last instruction, TOP would only capture the code for unpacking and the new code's semantics will not be consistent with our traces.

- **Indirect Symbolization.** At the data use point, after we determine the literal value to be a symbol address, we will look for the target instruction based on the PC in the taint record and rewrite the symbol. For instance, as shown in the second example in Table 3.1, at line 7 when "`call eax`" gets executed, we can infer that the value stored in `eax` is actually a function entry address. Since the taint record of `eax` is S[`eax`]=6, we will then rewrite the operand of the instruction at PC=6 and symbolize the literal value as `fun_0x409200`. For the return instruction, we will also check the taint record of the operand that is from the top of the stack. If it is tainted, we will update the target instruction as well. For instance, because line 4 "`push 0x401058`" and line 5 directly return, fetching value 0x401058 from the stack (this instruction is actually a case of control flow obfuscation), we will rewrite the operand of the `push` instruction as `L_0x401058`. Note that for a normal `call` instruction, unlike in this case, we will not taint its return address on the stack.

Case II: Global Data Symbolization. Compared with instruction operand symbolization, global data symbolization is more complicated and it requires not only translating the concrete global address into symbolic address, but also translating the *pointed* data stored in the global variable as symbolic. There are also two cases for global data symbolization depending on whether the global data accessed is implicitly initialized or explicitly initialized (or redefined).

- **Implicitly Initialized Global Data.** A global pointer (either data pointer or function pointer) could be initialized by a compiler, often with no more update during program execution. In this case, we need to symbolize the stored pointer value explicitly (with an assignment statement to be inserted at the beginning of the recovered code); otherwise the translated program will crash.

  Take the instruction at line 11: "`mov ebx, DWORD PTR [edi]`" as an example (the third row in Table 3.1). When this instruction is executed, we can infer that `edi` stores a pointer. At this moment, S[`edi`]=0x424a38, which indicates that we are dereferencing a memory address pointed to by the value stored in 0x424a38. In other words, we know that 0x424a38 actually stores a pointer and this pointer is implicitly initialized (because there is no other instruction to define this memory address). Therefore, we have to explicitly translate the content stored in 0x424a38. In this case, it happens to be 0x424a34. That is why we add an explicit assignment statement at line A in the last column (the address of this instruction depends on the final recovered code but it needs to inserted at the beginning of the recovered code).

  This example also indicates that *we need to track data-def and data-use of the memory cell* (details are elided since it is a standard algorithm). If there is no data-def for a particular memory cell, it will be a case of implicit data initialization. Similarly, for line 12 "`call DWORD PTR 0x424a3c`", we will add a function pointer initialization statement at line B in the last column, and that function pointer happens to point to a function at address 0x40102d. Similar to the statement in line A, this assignment statement should be placed at the beginning of the recovered code.

- **Explicitly Initialized Global Data.** If a global function pointer is explicitly redefined (e.g., memory `0x424a38` in line 13), its handling will be simpler. We only need to symbolize the address of the global variable, without adding any explicit assignment

statement (e.g., lines A and B in Table 3.1) for the global pointer variable. Instead, we will use the original program's code to dereference the memory. For instance, in the fourth example in Table 3.1, we only need to symbolize the operand at line 13 for memory addresses 0x424a38 and 0x424a30, and similarly at line 14 for memory address 0x424a38.

Again, we do not need to symbolize program heap and stack addresses. The main reason is that those addresses are dynamic hence the program code itself will initialize them and dereference them correspondingly. It is also important to note that *Memory Address Symbolization* is new in TOP; and none of the existing decompilation techniques, including Hex-Rays, has solved this problem, especially for global initialized pointer data.

3.2.3 Safety Instrumentation

TOP features the assurance of safety while leveraging the precise but incomplete dynamic analysis. To achieve this goal, we need to ensure that *the recovered code either behaves the same way as the original binary, or must throw predefined runtime safety exceptions.* We call this property *reuse safety*.

There are two root causes for safety violation: (1) A conditional jump (i.e., `jcc`) may not have both branches covered (as shown in Figure 3.1) and (2) an indirect `jmp/call` may not have all its targets covered.

- **Handling Missing Conditional Branches.** If only one of the branches is executed in a `jcc` instruction, TOP will generate a piece of exception handling code that will print a warning message and exit the program if the control flow of the recovered code falls through the missing branch. This warning message will be used to debug and refine the extracted code. In particular, we can leverage the input that induces the warning as an additional input to increase code coverage.

- **Handling Indirect Jumps.** Similar to conditional branches, indirect jumps/calls can lead to safety issues due to incomplete path coverage. Two types of indirect jump/call – `jmp/call` register and `jmp/call` memory – are considered. TOP translates an indirect jump/call into a list of conditional jumps with the possible values of the symbolized indirect jump/call operands as conditions. To guarantee safety, an assertion is added to the beginning of the conditional jump list to check if the target is among the symbolized addresses. In other words, our code creates a white list of the symbolized target addresses; any unknown (new) target will be captured and thrown out.

With the safety instrumentations above, we ensure that, when the recovered code is executed again, it will follow the original program's semantics or throw exceptions. Any attempt to execute a control flow path not executed before will be warned.

3.2.4 Instruction-to-C Translation

After we collect the program's CFG (Section 3.2.1), relocate concrete memory addresses to symbols (Section 3.2.2), and instrument the extracted program to ensure safety (Section 3.2.3), the next step is to translate the low-level instructions into C code. Since our targeted usage scenario is binary code reuse, users may not be interested in the implementation details of the extracted components but only their functionality – for example, a user may not care how a cipher function is implemented. Therefore, in the generated C code, we largely use the inlined assembly and their operands will be updated with symbolized addresses if any.

Also, since our goal is to recover the extracted code in the form of source code, we could generate the source code for instructions at various levels, including user level, library level, and even kernel level (if we use virtual machine-based dynamic instrumentation). However, we are most interested in the user-level code (because library code already exists). As such, the recovery process needs to stop when encountering a well-defined API. We also need to

dump the code from the beginning of the execution by default, or from the entry to the exit points specified by the user.

API Resolution. The goal of API resolution is to instruct our analysis to stop further tracing when entering the body of a library function and, at the same time, to use the well-defined interface and the API symbols to generate the function call (e.g., `malloc,printf,recv`). While there might be API-obfuscation in the binary code (e.g., to hide malware behavior from static analysis), our dynamic analysis can discover such obfuscation.

At a high level, our API resolution technique turns off address space randomization (which can be done by TOP users because they control the execution), and identifies the starting address for each library call. At runtime, we check whether the PC of an instruction matches any pre-defined API's PC. For Linux, we extract all the APIs in `glibc` and other dependent libraries if any, and resolve each API's symbol, arguments, and the starting PC.

For Windows binary, if DLLs are loaded at the virtual addresses specified as the image base address in a DLL's PE header[2], we will create a lookup hash table that contains all the virtual addresses of each exported API function from all known DLLs. If the calling target address can be found in the lookup table, we will retrieve the API calling context (i.e., name and arguments). If the DLLs are loaded to a nonstandard base address by system calls to explicitly map them to a different address space, we will look for specific calls to `NtOpenSection` which identifies the DLL name, and calls to `NtMapViewOfSection` which provides the DLL's base address. We then use the base address of this DLL to add those API addresses inside it.

Instruction Translation. Since TOP cares about code functionality more than code readability, we use a straightforward but effective inlined assembly rewriting approach to generate the corresponding C code. In particular, we traverse the BBHT (Section 3.2.1), starting from

[2]Since the bases of standard Windows DLLs do not conflict with each other, the loader and linker can load all DLLs at their specified base virtual addresses.

a user-specified address or a default address. For each executed instruction, if an absolute address is used, we will replace it with our symbolic address. If there is a missing branch that is not executed, we will instrument it with a safety check. If there is a global data access, we will associate it with an index (global variables are mapped to a big array) and initialize it with value in the memory. A control flow target will be rewritten as a specific symbolic label. If we encounter the end instruction, we will dump the C source code. Note that the C code is not just inlined assembly because we do recover information that is lacking in assemblies, such as variable types and function interfaces.

3.2.5 Offline Combination

To enable the *combination of multiple runs* of a binary, we also design a feature that serializes our in-memory meta-data to disk files, and then use the *offline-combination* component to combine the multiple runs for larger coverage. For offline combination, all the serialized meta-data are loaded into memory, and the CFG in the multiple meta-data is traversed. Whenever there is a path that is executed in one run but not in another, we will eliminate the safety instrumentation code and combine the two runs. After the combination, given the starting PC of a target function to reuse, TOP will traverse the BBHT, translate the instructions, and dump the source code reachable from the target function.

3.3 Evaluation

We have implemented a proof-of-concept prototype of TOP. To handle obfuscated Windows binary code, we choose QEMU-1.0.1 (Bellard, 2005b) as the underlying dynamic binary instrumentation engine. We perform virtual machine introspection (VMI) (Garfinkel and Rosenblum, 2003) to inspect the target process and thread, intercept all executed instructions, collect context information, and resolve and store the information in the BBHT. To

show the effectiveness and generality of TOP, we have implemented process/thread introspection and API resolution for both Windows and Linux. It is important to note that the five key components of TOP are *generic*, hence there is no need to customize for Windows and Linux.

More specifically, TOP needs to trace binary execution at thread level. To identify thread-level context, our introspection will use both process ID (by traversing the corresponding kernel data structure) and kernel stack pointer (with the lower 12 bits masked). This is because each thread will have a unique kernel stack (besides user level stack). Also, many programs create new processes. TOP tracks process creation by inspecting relevant system calls, and performs binary code translation and isolation for all child processes.

In this section, we present the results of evaluating TOP with a large number of legacy Linux binary programs (Section 3.3.1) and obfuscated Windows binaries (Section 3.3.2). The evaluation is performed on a machine with Intel Core i-7 CPU with 8GB physical Memory. The host Linux platform runs kernel-2.6.38; and the guest OS is Ubuntu-11.04 or Windows XP SP3.

3.3.1 Evaluation with Legacy Linux Binaries

Unlike the large pool of obfuscated binaries on Windows, much fewer obfuscated binaries exist on Linux. Hence our evaluation of TOP for normal, un-obfuscated binaries is mainly performed on Linux. We use the coreutils-6.11 package as our benchmark suite. Our goal is to use TOP to generate C code from execution traces, and further recompile them to generate customized binaries. We compare the functionality of the new binaries with the original ones for effectiveness evaluation.

Effectiveness. There are in total 100 programs in coreutils-6.11. Most of them are single process except three multi-threaded programs, `sort`, `mkdir` and `ginstall`. TOP can detect the thread-level control flow correctly. In our experiment, we compile them with "`gcc -O2`",

strip out their symbols, and run each of them to generate the corresponding source code from the trace.

When running these programs, we do not provide explicit command line option (e.g., we just type `ls`), unless we have to provide one such as for `cat`. For those that require files as input, we provide files with a size of 1KB. Next, we run the the recompiled binaries with the same option for tracing, and compare the output to test whether we retain the correct semantics of the original binary with the given inputs.

As expected, all 100 benchmark programs run successfully, and consistently generate the same result as their original counterparts for the same input. If we enter any other command line options, the new programs will generate exceptions and exit gracefully. Note that in such a case, the user can choose to further generate a more complete version of the recovered program using the exception-inducing input. For space constraint, we classify these 100 programs into seven categories and rank the source code size (in terms of LOC) generated for each program, as shown in the first column of Table 3.2. For each category, we report the top 3 programs in the 2^{nd} column of Table 3.2 for detailed presentation of our results. The 4^{th} column reports the number of symbolized memory addresses, and the 5^{th} reports the number of safety checks TOP added. These results reveal more details about the internal operations of TOP as well as the program-specific characteristics in these aspects. We observe that all programs require hundreds of symoblized addresses. For safety checks, the number varies across programs.

Performance Overhead. There are two kinds of performance overhead: (1) overhead of TOP tracing binary execution and generating new source code and (2) overhead of new software generated compared with the original software.

For the 100 programs, TOP on average runs about 0.2 second to generate the new program from a single trace. To further study performance overhead, we present the TOP runtime (incurred by its four online components) in the 6^{th} column in Table 3.2. For each program,

Table 3.2. Evaluation results with top-3 binary programs in 7 categories (in terms of recovered source code size).

Benchmark Programs			Online Phase of TOP			Offline Combination		Recovered Source Size (KLOC)	Consistent w/ Traces?
Category	Binary Programs	Assembly Code (LOC)	*#Symbolized* Addresses	*#Safety* Checks	Time (sec)	Test Cases	KLEE/TOP Coverage(%)		
Hash	sha512sum	25331	133	39	0.25	35	59.53	15.4	✓
	sha384sum	25331	133	39	0.25	45	59.74	15.4	✓
	sha256sum	9129	122	36	0.09	36	60.70	5.5	✓
File	vdir	18676	788	196	0.71	65	26.93	4.8	✓
	dir	18676	507	116	0.19	70	25.67	4.8	✓
	ls	18676	501	116	0.19	68	25.81	4.8	✓
Privilege	chown	10099	216	88	0.05	58	39.53	4.1	✓
	chmod	9811	208	81	0.02	54	29.77	2.9	✓
	chgrp	9436	196	80	0.04	50	31.08	2.9	✓
Disk	du	14887	487	173	0.06	66	38.43	5.7	✓
	df	9100	290	74	0.11	58	32.92	2.9	✓
	sync	4221	45	9	0.21	18	39.26	1.6	✓
Process	sleep	4541	85	25	0.08	58	42.05	1.9	✓
	kill	4935	83	31	0.04	37	38.71	1.9	✓
	nice	4680	43	11	0.05	47	41.17	1.9	✓
Environment	who	6097	363	147	0.20	27	36.34	2.2	✓
	env	4292	53	13	0.09	22	33.01	1.4	✓
	printenv	4290	50	9	0.09	12	24.08	1.0	✓
Utility	od	10172	245	61	0.09	119	53.42	5.5	✓
	pr	7611	480	130	1.01	79	40.03	3.0	✓
	wc	5959	210	45	0.24	45	44.47	2.7	✓

it takes less than one second for TOP to finish, except utility `pr` that takes 1.01 seconds. For each new program generated, we recompile and run it using the same arguments as those used for tracing. Since the translated code is almost identical to the original assembly code, the new program incurs negligible performance overhead, thanks to the use of inlined assembly.

Coverage. Recall that TOP supports combining multiple traces into one program (Section 3.2.5). To evaluate this feature, we use KLEE (Cadar et al., 2008) to generate test cases as inputs for program tracing by TOP. To simplify the experiment, we use the same KLEE command for coreutils programs.[3] The 7^{th} and 8^{th} columns of Table 3.2 show the number of test cases and the coverage for each program (use `klee-stats` command), respectively. The final source code size of these programs is reported in the 9^{th} column of Table 3.2. Meanwhile, the average offline combination time (not shown in Table 3.2) is 0.18 second.

3.3.2 Evaluation with Obfuscated Windows Binaries

Obfuscated binaries, especially those of malware, abound on Windows. The goal of binary code obfuscation is to disrupt analysis of the code and deter reverse engineering efforts. In general, there are three types of widely used binary analysis platforms: disassembler, debugger, and virtual machine (VM). Consequently, obfuscation techniques can be categorized into anti-disassembler, anti-debugger, and anti-VM. For each category, there exist a variety of techniques. For example in the anti-dissembler category, there exist the techniques of garbage code insertion, control flow obfuscation, instruction aliasing, binary code compression, and encryption.

To evaluate the resilience of TOP against these obfuscation techniques, we select 15 representative obfuscated samples from (Branco, 2012) (shown in Table 3.3), which cover

[3] http://ccadar.github.io/klee/CoreutilsExperiments.html

the state-of-the-art obfuscation techniques. The samples and their source code, plus two additional binary-only, packed samples, are from *offensivecomputing.net*. Note that a reason for selecting the 15 samples is that they allow us to verify the recovered code's correctness. For the two packed malware samples, we have no knowledge about their implementation, except that they are packed.

Effectiveness. Interestingly, many of these binary samples have only 1536 bytes as shown in the 2^{nd} column of Table 3.3. The reason is that their source code is very small, each containing only a few lines of code for simple demonstration of an obfuscation technique. That also explains why the recovered source code is also small (shown in the 9^{th} column). For all the obfuscated samples, TOP successfully recovers their source code from traces. We have run the recovered and recompiled programs and verified that they have consistent semantics with the original binaries (last column).

To better illustrate the strength of TOP, consider the two pieces of assembly code in Table 3.4, one from the sample `middle_instruc tion.exe` for instruction aliasing, and the other from `garbage_ bytes.exe` for adding garbage code. For instruction aliasing, we see that the original code's execution at line 3 will jump to the middle of the instruction at line 1 as the condition for `jz` is always true (by `xor` in line 2). Then `eb 05` in line 1 would be disassembled as a direct `jmp` which goes to line 5. In other words, the garbage byte in line 4 will never be executed. However, IDA Pro cannot disassemble this obfuscated code because it would disassemble `db` as opcode `call` at line 5 in the second column. Due to incorrect alignment of instructions, the subsequent disassembling would be incorrect. In contrast, TOP dynamically generates the correct results as shown in the third column. Note that in line 6, TOP adds a safety guard (`jmp`) to handle partial coverage issue. For the second example, since a garbage byte `0x6a` in line 3 is introduced, IDA Pro fails to perform static disassembling. In both cases, we can compile the code generated by TOP.

Table 3.3. Evaluation results with 17 obfuscated binary programs.

Obfuscated Samples	#bytes	Obfuscation Techniques	Anti-Disas.	Anti-Debug	Anti-VM	#Symbolized	#Safety	#LOC	Coverage	Consistent?
garbage_bytes.exe	1536	Adding garbage bytes	✓			3	1	38	71.4%	✓
program_control_flow.exe	1536	CFG Obfuscation (push/jmp)	✓			3	0	30	100%	✓
pushret.exe	1536	CFG Obfuscation (push/ret)	✓			3	0	40	100%	✓
call_trick.exe	1536	CFG Obfuscation (call/push/ret)	✓			8	0	71	100%	✓
middle_instruction.exe	1536	Instruction Aliasing	✓			10	1	78	100%	✓
Win32.Bamital.exe	22016	Encryption Packer	✓			303	37	1264	-	✓
Virus.Win32.Adson.exe	5632	Compression Packer	✓			60	6	739	-	✓
hardware_bp.exe	1536	Hardware Breakpoint		✓		12	4	78	70.8%	✓
heapflags.exe	1536	Heap Flags Detection		✓		9	1	59	81.8%	✓
instr_counting.exe	1536	Instruction Counting		✓		12	3	136	65.6%	✓
ntglobal.exe	1536	PEB NtGlobalFlag		✓		9	1	71	81.8%	✓
peb.exe	1536	IsDebuggerPresent		✓		9	1	59	81.8%	✓
rdtsc.exe	1536	RDTSC Instruction Timing		✓		8	1	75	90.0%	✓
softice.exe	1536	Softice Interrupt		✓		8	1	75	83.3%	✓
software_bp.exe	1536	Soft Breakpoint Detection		✓		10	1	77	91.7%	✓
ss_register.exe	1536	SS Register		✓		9	1	83	88.2%	✓
anti-vm_in_instruction.exe	1536	Anti-Vmware IN Instruction			✓	7	0	59	100%	✓

Table 3.4. Disassembling results from IDA Pro and TOP for obfuscated programs.

Original Assembly Code	Disassembly from IDA Pro	Disassembly from TOP
1 66 b8 eb 05 mov ax,0x05eb 2 31 c0 xor eax, eax 3 74 fa jz $-4 4 e8 db 0xe8 ;garbage byte 5 58 pop eax (middle_instruction.exe)	1 loc_401006: 2 mov ax, 5EBh 3 xor eax, eax 4 jz loc_401008 5 call near ptr 6A98686Bh	1 mov ax, 0x5eb 2 xor eax, eax 3 jz loc_0x401008 4 jmp loc_ERROR 5 loc_0x401008: 6 jmp loc_0x40100f 7 loc_0x40100f: 8 pop eax
1 31 c0 xor eax, eax 2 74 01 jz .destination 3 6a db 0x6a ;garbage byte 4 .destination: 5 58 pop eax (garbage_bytes.exe)	1 xor eax, eax 2 jz loc_401007+1 3 loc_401007: 4 push 58h	1 xor eax, eax 2 jz loc_0x401008 3 jmp loc_ERROR 4 loc_0x401008: 5 pop eax

Performance overhead. With the small size of the majority of these samples, TOP quickly recovers their source code within several milliseconds. For `Win32.Bamital.exe` and `Virus.Win32. Adson.exe` which are real-world malware, it takes TOP about 14 seconds to perform online tracing. The reason for the much longer time is that both malware binaries involve many iterations for decryption and decompression.

Coverage. Unlike the Linux samples, we do not have KLEE to generate test cases for the obfuscated binaries to improve coverage. Instead, we perform a manual check on the coverage for the obfuscated samples with source code, as reported in the 10^{nd} column. The TOP-generated code has high coverage, with 5 of them having 100% coverage. For the two malware samples, we do not estimate their coverage due to lack of ground truth.

3.4 Summary

We have presented trace-oriented programming (TOP), a new framework to enable the reuse of legacy binary code from execution traces. Through dynamic execution of a binary, TOP collects necessary information such as control structures, memory addresses and accesses, and safety information; and then translates each executed instruction into a predefined

template or inlined assembly according to its semantics. While TOP shares the same goal with existing decompilation techniques, it enjoys unique benefits from dynamic analysis, such as being obfuscation resilient and free from point-to analysis. We have implemented a proof-of-concept TOP prototype. Our evaluation results with over 100 legacy binaries (including malware binaries) indicate the effectiveness, efficiency, and safety of TOP and demonstrate the application of TOP to malware analysis and security function reuse.

We have implemented a TOP prototype and evaluated it with a range of benign and obfuscated binary programs. Our evaluation results show that we can directly compile the reconstructed source code into new binaries, and run them correctly with the functionality we have traced. Using TOP, we will demonstrate two compelling applications: (1) malicious code inspection and identification, and (2) security function transplanting in Chapter 6.

In summary, we makes the following contributions for TOP:

- We present trace-oriented programming (TOP), a new approach to reconstruct program source code from execution traces. Unlike decompilation that statically transforms a piece of binary code, TOP dynamically translates it with more runtime information and directly generates reusable software components.

- We devise a number of enabling techniques for TOP such as program control structure recovery, address symbolization, safety instrumentation, and instruction-to-C translation. These techniques work together to recover the source code from dynamic execution of a binary, and ensure that the recovered code has well-defined behavior consistent with the execution traces.

- We propose a systematic technique to symbolize the function pointers that are initialized and in global memory regions. To the best of our knowledge, this has not been proposed in any existing static decompilation technique including Hex-Rays.

- We have built our prototype systems for Windows and Linux platforms, and tested them with over one hundred pieces of benign and obfuscated binary code. In addition, we apply TOP to two security applications: malware analysis (e.g., unpacking) and identification, and security function transplanting.

CHAPTER 4

REFIT: RECOVERING OF FUNCTION INTERFACE FOR REUSE[1]

In this chapter, we present in details our second component (REFIT) which is able to uncover function interfaces. In particular, we begin with technical overview in Section 4.1, then detailed design in Section 4.2 and evaluation in Section 4.3, and end with summary in 4.4.

4.1 Technical Overview

We present REFIT, which aims at precisely *RE*covery of *F*unction *I*n*T*erfaces. More specifically, given a piece of application binary code (including the dynamic shared library), REFIT will automatically identify the function interfaces of all the involved functions (or function of interest depending on analyst's specification). This function interface includes the number of parameters, the syntactic type and the semantic type for each parameter and return value, and the program state affection of the function (e.g., the global variable updated, and the heap changes).

In this section, we define our research problem and outline our approach. We first identify the challenges in our function interfaces recovery from legacy software in §4.1.1, then formally define our research problem in §4.1.2, and finally give an overview of our techniques in §4.1.3.

4.1.1 Challenges

The goal of REFIT is to precisely recover the function interface (including the number of implicit and explicit parameters, and their corresponding syntactic and semantic types) such that programmers can reuse the binary functions (e.g., if the function is located in a

[1]This chapter contains material that is still under submission at the time when writing this dissertation.

library/DLL, a programmer can then call this function once its interface is clear; if the function is in user space, then programmers can extract the binary code for the reuse (Caballero et al., 2010; Kolbitsch et al., 2010; Lin et al., 2010b; Dolan-Gavitt et al., 2011)), or reverse engineer analyst can better understand the internals of the software. To this end, we have to address the following challenges:

C1. Parameter Location Identification. Unlike in high level programming language (such as C), the parameter passing is hard to be recognized at binary code level. Intuitively, `PUSH` instruction maybe a natural indication for passing parameters. However, besides `PUSH`, there are many other instructions that can also be used for parameter passing, such as `MOV`, `FST` and `FSTP`.

Moreover, besides passing arguments, many of these instructions have other semantics: they may act as a normal data movement. For instance, as shown in Figure 4.1, an instruction snippet of code from `ls` program (coreutils-8.15), normal data movement and parameter passing both use `MOV` instructions. In particular, notice that function `format_user` invokes `getuser` in Figure 4.1(a). Function `getuser` takes one parameter; correspondingly, its assembly code should have one data movement instruction to pass parameters (line 1 in Figure 4.1(b)). However, line 2 can be misclassified as parameter passing, if we simply look at the instruction patterns. In other words, we need techniques to precisely recognize where the arguments are.

C2. Non-trivial Parameter Type Identification. In order to reuse the reverse engineered functions, both the syntactic type (i.e., the size and offset for each field) and semantic (i.e., the meanings) type of the parameters should also be recovered. Consider function `Blowfish_Encrypt` in Figure 4.2, if the shape for `BLOWFISH_CTX` is not recovered (and then if some elements are not allocated), any accesses to its element (line 312) would crash the reused program.

```
3657 static void
3658 format_user (uid_t u, int width, bool stat_ok)
3659 {
3660  format_user_or_group (! stat_ok ? "?" :
     (numeric_ids ? NULL : getuser (u)), u, width);
3661 }
```

(a)

```
0804ca00 <format_user>:
              ...
1 804ca28: mov    %ebx,(%esp)
2 804ca2b: mov    %edx,0x1c(%esp)
3 804ca2f: call   8054580 <getuser>
              ...
```

(b)

Figure 4.1. Challenges in parameter recognization.

```
10 typedef struct {
11  uInt32 P[16 + 2];
12  uInt32 S[4][256];
13 }BLOWFISH_CTX;


301 void Blowfish_Encrypt(BLOWFISH_CTX *ctx,
                         uInt32 *xl, uInt32 *xr){
  …
311   for (i = 0; i < N; ++i){
312      Xl = Xl ^ ctx->P[i];
  …
317  }
   …
}
```

Figure 4.2. Non-trivial function interface case.

However, recovering the data structure type is challenging, especially with complicated data structures such as the aggregated data types (e.g., arrays and structures) and the recursive types (e.g., linked lists and trees). Thus, we have to identify the layout of the aggregated type data structure and make sure the correct offset for each element in the structure. In addition, if there is any recursive data structures, we need to further identify their shapes.

C3. Extracting External Data Dependency.. Not all the functions are self-contained, and many useful functions often have external dependencies. In general, data accessed in a function can be classified into internal (function local variables) and external (function arguments, heap variables and global variables). Thus, we need a mechanism to identify these external data. For external heap variables, we need to identify their shapes (in general they are just directed graphs) and abstract their types.

C4. Static vs. Dynamic.. There are always tradeoffs between static and dynamic analysis: static analysis is less precise but complete, whereas dynamic analysis is vice-versa. In binary code reuse, we do not want the reused program crash and we care more about the preciseness, especially when facing with pointers. As such, we would like to use dynamic analysis since it is points-to/alias analysis (Burke et al., 1995; Liang and Harrold, 1999; Pearce et al., 2007; Debray et al., 1998; Brumley and Newsome, 2006) free (i.e., points-to relations are exercised in execution).

4.1.2 Problem Statement

Recovery of function interfaces needs to solve three problems:

- **Discovering function protocols.** We need to discover the input type and output type of the function of interest, which includes the number of implicit and explicit parameters, and their syntactic (layout and shape) and semantics (meaning) types.

Using dynamic analysis, we are only able to observe the values. Therefore, we have to translate the concrete values into abstract types (e.g., `int_t`, `float_t`).

- **Annotating the types with syntactics.** It is not sufficient to only discover the layout and field types. We have to further annotate the syntactic types, especially for recursive data structure types. For instance, during the binary code execution, we are able to observe pointers, but we would like to know a pointer is a linked list pointer (e.g., `next_t *`), a tree pointer (`tree_ left_child_t`), etc.

- **Annotating the types with semantics.** Semantic meaning of a data field plays the key in program understanding and binary code reuse. At binary code level, we are able to observe the OS level and standard API level semantic meanings (e.g., `pid_t`, `sockaddr_in`, `FILE*`), we have to name the field with the recovered semantic meanings.

4.1.3 Architecture Overview

An overview of REFIT is presented in Figure 4.3. There are three key components in REFIT: (1) *(Implicit, Explicit) Parameter Discovery*, (2) *Online Semantic Type Recovery*, and (3) *Dynamic Shape Analysis*. Given a piece of input binary and test suits, along with the entry point of the functions of interest (or all the functions involved by default), REFIT will recover the function protocols (i.e., the interfaces) including the number of the arguments and return values, as well as their specific syntactic and semantic types.

While binary code extraction is not our focus in this chapter, for a complete demonstration of our techniques, we also integrate the binary code extraction approach proposed in BCR (Caballero et al., 2010) that disassembles and translates the binary code into inlined assembly.

Assumptions. To narrow down the scope of our problem, we assume the application binary code is running on top of 32-bit x86 architecture, and is generated from C source code. Also,

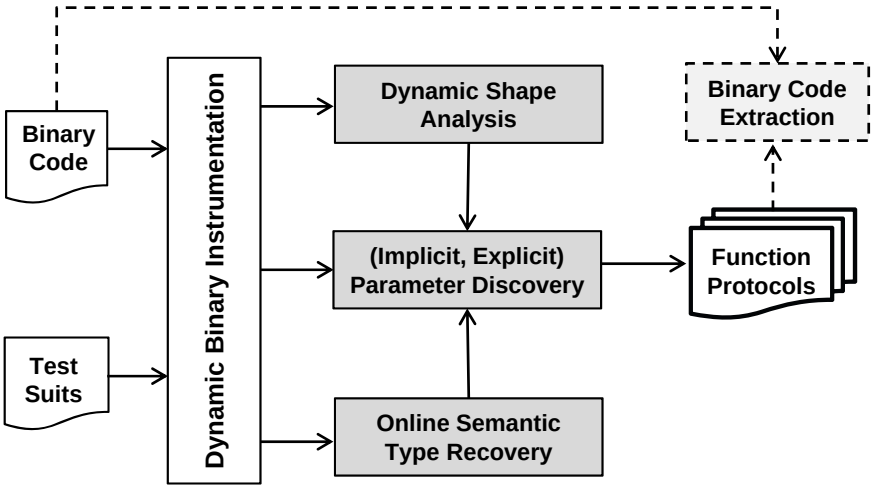

Figure 4.3. Overview of REFIT.

we assume we know the entry point of the to-be-reused function, and this knowledge can be acquired through such as dual slicing (Weeratunge et al., 2010) or differential slicing (Johnson et al., 2011). Our goal is to automatically find its closure and enable the reuse of the function code of interest. In addition, we assume there is no obfuscation and the calling convention is `cdecl` that originates from the C programming language and is used by many C compilers for the x86 architecture (de Boyne Pollard, 2004). Note that in `cdecl`, subroutine arguments are pushed on the stack in the reverse order (as shown in Figure 4.4), and integer values and memory addresses are returned in the `eax` register.

4.2 Detailed Design

In this section, we present the detailed design of each component of REFIT. We first present how we recover the parameters in §4.2.1, then their semantic types in §4.2.2, and finally describe how we perform the dynamic shape analysis in §4.2.3.

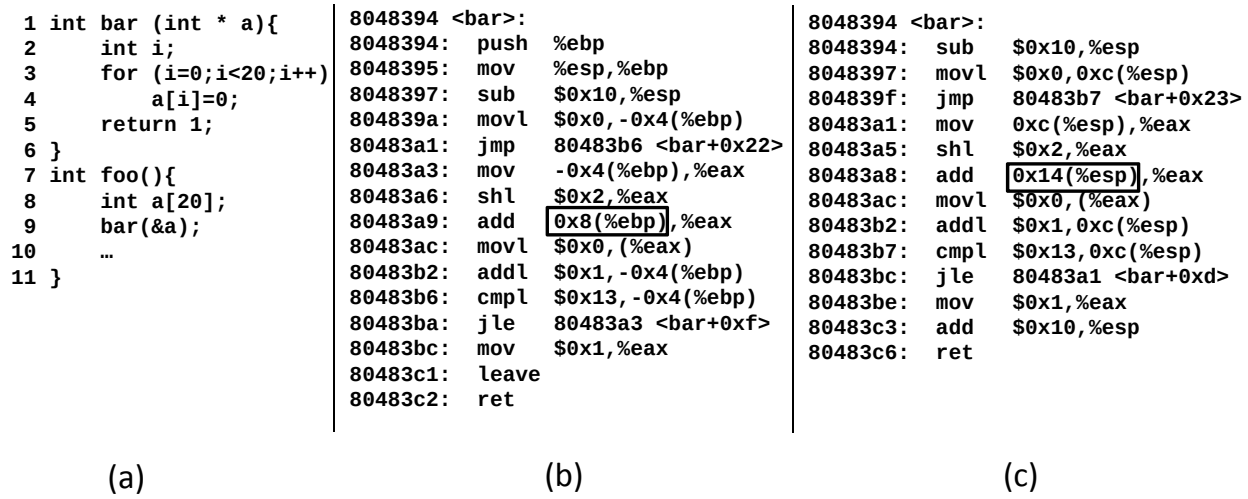

Figure 4.4. Examples on argument accessing inside a function.

4.2.1 Parameter Discovery

Intuitively, any data passes through the boundary between inside and outside of a function should be considered as a parameter. Therefore, parameter discovery problem is essentially a data dependence problem, and we should track whether or not the data accessed inside a function depending on any outside data. It might be trivial for global and heap data, as their definitions can be easily identified. The only challenge lies in stack data because of the parameter passing.

Observation. In x86, the explicit function parameters are often stored in the memory region whose address is greater than the function local variables. The boundary between parameters and local variables is a memory cell storing the function return address. Before the execution of the first instruction (typically `push ebp` to store the caller's stack frame pointer if it is not omitted, as illustrated in Figure 4.4(b)) of a function, the return address is pointed by register `esp`, and we call this `esp` as the function entry point `esp` (denoted as `ESP`).

Since a function can be called multiple times by different functions, the location of its argument and local variables cannot be statically determined. As such, to access the pa-

rameters (or local variables), the only way is to derive the address from the `ESP` (because all the function call must have a return address stored in the stack in x86). During the execution of a function, `ESP` can be propagated to many other registers such as `ebp` (Note that when stack frame pointer is not omitted, register `ebp` will usually store the `ESP`) or `edi`. Therefore, more registers would become a stack frame pointer, like `ebp`. We have to identify all the stack frame pointers by tracking the propagation of `ESP`. That is, we have to perform a dynamic taint analysis that has been widely used in many applications (e.g., (Chow et al., 2004; Newsome and Song, 2005; Egele et al., 2007; Lin et al., 2010a; Slowinska et al., 2011)) of the `ESP`. The function argument will be determined based on whether the address is greater than `ESP` and its base address register is derived from `ESP`. For instance, `0x8(%ebp)` in Figure 4.4(b) and `0x14(%esp)` in Figure 4.4(c) will be classified as function parameters of `bar`.

One might wonder whether we can directly use a memory address range to determine the arguments (instead of using taint analysis of `ESP`). For instance, one could have a heuristic that during the execution any address greater than `ESP` can be classified as arguments. However, as shown in Figure 4.4, the local variables of function `foo` will also be classified as arguments if we only compare the address ranges.

Then the next question is what we should use to represent the taint record. Intuitively, a taint bit might suffice to represent whether a memory access has a base address derived from `ESP`. However, as a function can call another function which can further call other functions, we have to differentiate the parameter for each function. Thus, our taint record for parameter identification is actually an unsigned integer value. This value can directly pin-point which stack frame is for a given argument address access.

In addition, there are also implicit parameters. For instance, a function can directly use a register to pass a parameter, and we have to detect register-based parameter passing as well. Our observation is if there is any access to an uninitialized register inside a function execution, this register is a parameter. Meanwhile, a function can also directly access program

global variables. Fortunately, memory addresses of global variables can be pre-determined by parsing the ELF binary image files (as our input binary is not obfuscated).

The Algorithm. Our parameter discovery algorithm is based on dynamic taint analysis. Since this technique has been widely investigated, we will not present its technical details and instead we will particularly describe how we customize it for our purposes.

At a high level, for register-based parameter passing, our algorithm is to detect whether a source register is used without initialization inside a function; if so, it is a passing parameter by register case. For each memory accessing inside a function, we detect whether the memory location is a function parameter based on the taintedness of its base address. Specifically, if the base register for an indirect memory access is a stack frame pointer and if the memory address is located above `ESP`, we detect it as a parameter, use the taint value to determine which function in call stack a memory location belongs to and extract the entry `ESP` for that function, and set up their parameters. In addition, we detect their accesses based on the address ranges (e.g., the addresses falling into `.data`, `.bss`, `.rodata`, `.got`, etc.).

4.2.2 Online Type Recovery

Next, we describe our online semantic type recovery for function parameters. While our main goal is to recover types for function parameters, in fact our algorithm can (and need to) type other memory locations in stack, global and heap memory because of their correlations with parameters. In other words, we have to perform a general semantic type inference.

Observation. Recently, a number of type inference techniques were proposed. They can be classified into dynamic approaches (e.g., (Guo et al., 2006; Lin et al., 2010a; Slowinska et al., 2011)) and static approaches (e.g., (Balakrishnan and Reps, 2004, 2007; Reps and Balakrishnan, 2008; Lee et al., 2011)). As Refit is dynamic binary analysis based, the most closely related work is our REWARDS (Lin et al., 2010a). However, REWARDS has to log the execution trace to facilitate the offline inference to eventually recover the semantic types.

This offline based approach usually consumes a large volume of resources. To advance the-state-of-the-art, we propose a new efficient, online algorithm to perform the type inference, and this algorithm is inspired by both REWARDS (Lin et al., 2010a) and aggregate structure identification (Ramalingam et al., 1999).

In particular, since data types are essentially abstractions related to what to store and how to update memory, we associate a shadow record for each memory cell to keep track of their types. The difference compared to REWARDS is how many shadow records we need to use. In REWARDS, each shadow record is associated with each *dynamic* memory address, and the shadow record has to store the types and timestamps. Also, REWARDS involves an offline analysis to infer types. In REFIT, we associate a shadow record to an abstract memory that corresponds to multiple *dynamic* addresses. As such, we require less space and also eliminate the need of an offline analysis as in REWARDS.

Our Approach. However, the mapping between dynamic address and abstract address is non-trivial as the mapping is usually multiple-to-one, for instance multiple dynamic heap addresses can be mapped to only one static heap pointer variable. In the following, we present how we construction such mappings.

- **Global.** Global addresses are static. Hence, the abstraction of a global address is the address itself. We allocate a big shadow record array for all the global variables. The address range of global addresses (i.e., the size of our array) is acquired by parsing the ELF files (e.g., the address range of `.data,.bss` section). Upon a global memory access, its shadow record can be looked up by computing the proper offset.

- **Heap.** Heap addresses are entirely dynamic. We use the program counter (PC) of the caller of the `malloc`-family functions, along with the call chain as the abstraction, denoted as a $HeapID$. That is, all heap objects allocated at the same PC, through the same call chains are considered having the same type. $HeapID$ is used to identify heap

object types especially for recursive data structures. We use the offsets to index the individual fields in each allocated heap object. To speed up the lookup from a dynamic heap address to its abstraction, we use a red-black tree to track all the allocated heap objects and maintain the mapping.

- **Stack.** Local variables are allocated on stack. Hence, depending on the calling context, the same local variable may have different addresses. Hence, we consider the run time stack frame as a local array for one particular function. As illustrated in Figure 4.5, the size of the local variables can often be determined by instrumenting instructions such as `sub $imm,esp`. Therefore, similar to global variable recoveries, a concrete stack address can be abstracted to an offset index within the local array of the function. However, a caveat is that function parameters are also located in the stack frame. The space of parameters may be allocated together with local variables (by the same `sub $imm,esp` instruction) as shown in Figure 4.5(a), or by separated `push` instructions as the case shown in Figure 4.5(b). Hence, there is no obvious pattern to determine the memory range for locals. Fortunately, our previous parameter location identification component can be leveraged.

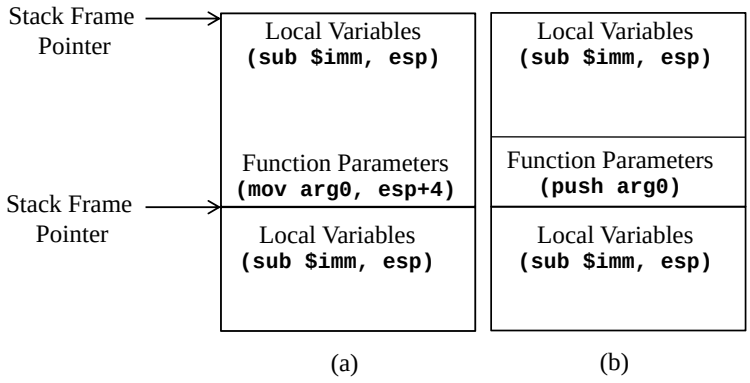

Figure 4.5. Locations of Function Parameters.

As discussed, our shadow records will store information to facilitate type inference. The basic type inference is at type revealing instructions (e.g., the arguments of a system call), whose operand types can be resolved, we resolve the types. For data flow propagation instructions, we perform type unification (Guo et al., 2006; Lin et al., 2010a; Ramalingam et al., 1999). As there are primitive types (e.g., `int32`, `char`) and composite types (e.g., structure, array). In the following, we present how we resolve these data types in greater detail.

Primitive Type Resolution. Primitive types can be discovered by directly inferring the x86 instruction semantics. For instance, float point instructions would be always operating with float point types. Since type will be flowed (by following the data flow), certainly we need to perform type unification when two variables merge. However, during type unification, there might be type conflicts, e.g., a 32 bits integer could be `uint32` or a data (code) pointer. To resolve such conflicts, we define a type lattice to guide our type unification. The lattice for primitives is presented in Figure 4.6. In the lattice, $\top$ represents an unknown type, and $\bot$ represents a type conflict. All types are connected to $\bot$. If there is any type conflict for these primitive types, we will choose their unification as their new type, and our goal is to infer the refined primitive types as far down the lattice as possible without reaching $\bot$.

Composite Type Resolution. We also need to recover aggregate types, which are often composed by primitive types. In particular, we need to recover the size, offset (i.e., the layout), and the primitive type for each field inside a structure or array. This is crucial especially for pointer fields. If we make mistakes when passing data structures with pointer fields as parameters to the reused code, we would easily crash the reused program.

To recover composite types, we need to classify memory based on their regions. For global variables, we consider them as an array, and the size is determined from header sections of the executable files. If there is any pointer pointing to other locations, their type will be resolved based on their target. For heap variables, their allocated size retrieved from the

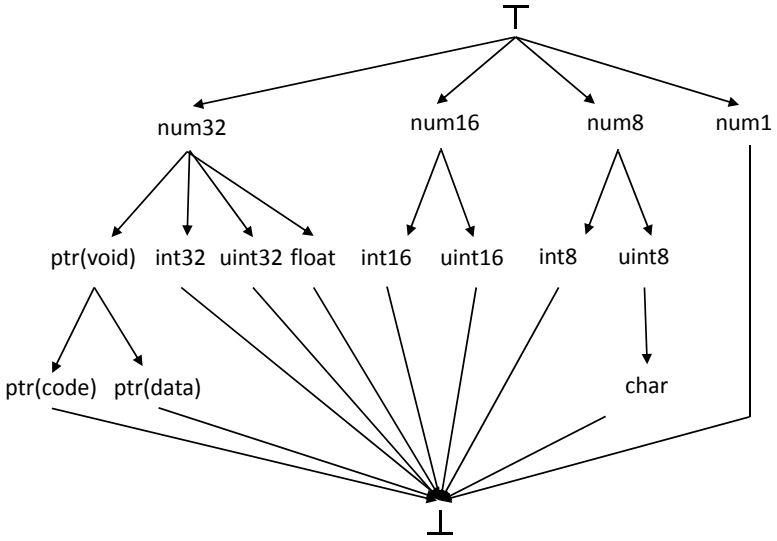

Figure 4.6. The Primitive Type Lattice in REFIT.

`malloc`-family functions will be the heap object size, and the type of a particular heap object is based on the $HeapID$. To identify recursive data structures such as link list, we would observe a pointer pointing to the same $HeapID$ inside a heap object of this particular type (direct recursive data structure) or its children has pointer pointing to itself (indirect recursive data structure). For local variables, the allocated stack frame will be their size.

Semantic-Type Identification and Annotation. In the reuse scenario, end-users often need to know the meanings of the arguments. For instance, our primitive type may resolve a `pid_t` type with an integer type, but if we know further that integer is actually a `pid_t`, which would be better. In other words, we would like to recover the semantics of the data structure field, and annotate the field name with the semantic information. For the OS or library level semantics of the data structure field (e.g., `pid_t` or `ipaddr_t`), we use the same approach as in REWARDS (Lin et al., 2010a) to resolve them.

Shadow Record Design. Each translated static memory address at byte granularity has a corresponding shadow record to track its type information. We do not directly store the resolved concrete type in the shadow record, and instead we store a pointer that points to

a record that contains (1) the resolved type for this record if the type has already been resolved, or $\top$, (2) the $HeapID$ if this memory cell has a pointer pointing to a heap object.

By introducing this one-layer indirection, we eliminate the offline backward type inference in REWARDS. More specifically, when a type revealing instruction resolves a type, REWARDS will visit the log file to resolve the previous visited data with the same type. In REFIT, when a new type is resolved, all other static address connected to this type is automatically resolved. That is, there is no backward type resolving anymore.

4.2.3 Dynamic Shape Analysis

Observation. Only identifying the types (e.g., the layout, primitive and composite type of the data structure) is still insufficient, especially for pointer type field that points to itself (i.e., the recursive type data structure as illustrated in Figure 4.7). For instance, suppose a data structure has two pointer fields that both have the same $HeapID$, we still do not know whether this data structure is a double linked list, a tree, a directed acyclic graph (DAG), or just a directed graph (DG). Consequently, end users still do not know how to use this data structure when preparing for a parameter with such a type, and they have to know the shapes of the data structure.

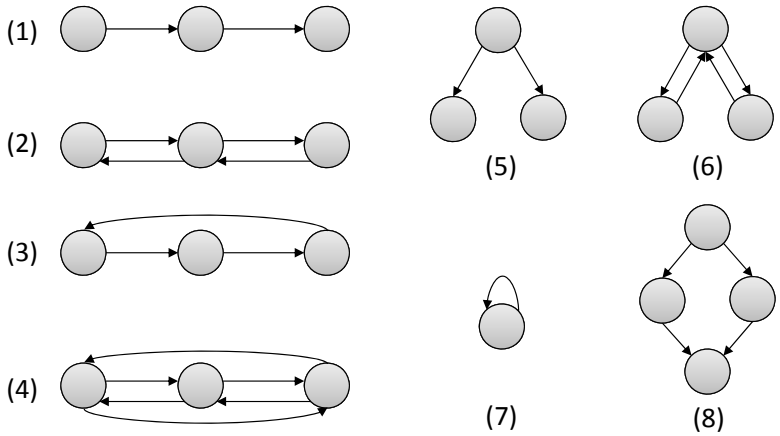

Figure 4.7. Different Shapes of the Recursive Data Structures.

Our Approach. To infer the shapes of the recursive data structures, earlier approaches have been using abstract interpretation (Fachbereich, 1999) or other region based approaches (e.g., (David R. Chase and Zadeck, 1990; Alain Deutsch INRIA Rocquencout, 1994)). In our particular domain, we adopt the region-based approach and leverage the runtime instances of the data structures, to build the data structure instance graph $G = \langle V, E \rangle$ where V denotes the vertex set (i.e., $v_i \in V$) of G, and E denotes the edge set (i.e., $e_i = \langle v_k, v_j, o \rangle \in E$ where o is the pointer field offset inside the data structure) of G, from program execution, and identify the shapes by traversing G.

As a result, shape analysis becomes an edge classification problem in the data structure instance graph. For a particular recursive data structure, there must exist an edge-invariant for the pointer field. For instance, the `next` pointer in a linked list would always have the `next` pointer type, which is an invariant. Our goal is to identify such invariant. Based on the instance graph and different graph edge definitions (Cormen et al., 2009), we perform a depth-first-search (DFS) starting from parameters (the root of the DFS traversal), and classify each edge $e_i = \langle v_k, v_j, o \rangle$ visited, and then identify the edge-invariant. The edges of our interest include:

- *Tree edge*: if v_j was first traversed by exploring edge $\langle v_k, v_j, o \rangle$ in DFS.
- *Back edge*: if v_j is an ancestor of v_k in DFS.
- *Forward edge*: if $\langle v_k, v_j, o \rangle$ is *Non-Tree-Edge*, and v_j is a descendant of v_k in DFS.
- *Cross edge*: all other edges in instance graph.

Once we have classified each edge in DFS tree, we perform an invariant discovery by checking each specific offset, the corresponding edge type. More specifically, we identify the following recursive field based on their edge invariant.

1. Linked list `next`: if all the edges of G at the offset o_f are *tree edges*, this edge is a link list `next` edge-invariant, and we recover this pointer type as linked list `next`. If there is only one such pointer field inside this data structure, we will further recover it as singly linked list `next`.

2. Doubly linked list `next` and `prev`: if G contains a *tree edge* at offset o_f and $\forall\langle v_i, v_j, o_f\rangle$, $\exists\langle v_j, v_i, o_b\rangle$, fields at o_f and o_b both have invariant, and we recover o_f and o_b as the *next* and *prev* of a doubly linked list, respectively.

3. Circular singly linked list `next`: if G contains both a *tree edge* and a *back edge* at offset o.

4. Circular doubly linked list `next` and `prev`: if G contains both a *tree edge* and a *back edge* at offsets o_f, and $\forall\langle v_i, v_j, o_f\rangle$, $\exists\langle v_j, v_i, o_b\rangle$, we recover o_f and o_b as the *next* and *prev* of a circular doubly linked list, respectively.

5. Tree `child` pointer: if all edges in G are *tree* edges.

6. Tree with `parent` pointer: if G contains *back edge* at offset o_p, all other pointers are only seen on *forward* edges, and for all edges $\langle v_i, v_j, o_p\rangle$ there exists a forward edge $\langle v_j, v_i, o\rangle$. We recover the pointer at o_p as the `parent` pointer in the tree and all other as the `child` pointer.

7. Self pointer: if G contains a *back* edge to itself (i.e., $\langle v, v, o\rangle$).

8. DAG pointer: if G contains *cross* edges, but there are no *backward* edges.

These pointer shapes will guide end-users to prepare for the parameters. If the pointer cannot be recovered further by the above invariant-assisted shape analysis, we will just output their identified data structure type.

4.3 Evaluation

We have implemented our REFIT using PIN-2.10 (Luk et al., 2005) with over 7K lines of C code (LOC). In this section, we present our experimental result. We first tested its effectiveness using a number of binary programs from Linux platform in §4.3.1. Then we describe how we reuse these functions in §4.3.2. The configuration of our testing platform is Intel (R) Core (TM) i7-2600 CPU with 4G RAM, running Linux kernel 2.6.38-15 (32-bits).

4.3.1 Overall Results

To evaluate the effectiveness of REFIT, we have to first select the benchmark programs and target functions. Those selected as benchmark programs have the following criteria: (1) have logically independent functionality and thus can be reused as a component in new software development; (2) are algorithm-oriented such that it is hard for developer to reimplement; (3) have high coverage for each execution trace. As such, in our evaluation, as shown in the first column of Table 4.1, we selected 7 open source programs from hash, cryptography, sorting, random number generation, compression and decompression. The source code is only used to verify the correctness of our result. For each of the testing program, our goal is to identify the function interfaces, and extract and reuse them in new software. In 2^{nd} column, for each of the binary program, we report the number of user-defined functions dynamically executed (D). To get a sense of the coverage, we also reported the total number of functions statically involved in the user-defined functions (S). And in 3^{rd} column, the analysis time spent for each benchmark program in the last column is reported.

Table 4.2 shows the overall statistics for these 7 programs, regarding the accuracy of parameter discovery, semantic type recovery, and syntactic type recovery. By default, we use REFIT to recover the interfaces of all of the executed functions. We collect the ground truth of these functions by instrumenting LLVM (especially `Module::iterator` and `Function::arg_iterator`), and then cross-check the result.

Table 4.1. Summary of the Benchmark Programs

Binary Programs	Category Hash	#Function (D/S)	Time (Sec.)
sum	Cryptography	13/79	8.17
bcrypt-1.1	Compression	22/26	17.61
tsort	Sorting	23/82	10.22
bsdsort-1.0	Sorting	9/39	8.25
ncompress	Compression	4/12	5.65
r250	Number	5/7	48.52
quasi	Number	5/5	6.41

Table 4.2. Statistics of the Overall Result of the Benchmark Programs

Binary Programs	Parameter Discovery				Syntactic Discovery			Semantics Discovery		
	Global	(R/T)	FP	FN	(R/T)	FP	FN	(R/T)	FP	FN
sum	1.46	2.63/2.75	4.34%	4.34%	1.06/1.06	0.00%	0.00%	1.06/1.13	0.00%	4.34%
bcrypt-1.1	0.18	3.90/2.85	15.38%	0.00%	2.10/2.20	0.00%	7.04%	1.00/1.10	0.00%	3.19%
tsort	1.04	1.86/2.05	0.00%	3.84%	1.33/1.33	0.00%	0.00%	0.95/1.00	0.00%	3.84%
bsdsort-1.0	6.44	3.67/3.95	0.00%	8.69%	1.86/1.95	0.00%	3.61%	0.86/0.86	0.00%	0.00%
ncompress	9.75	1.57/1.57	0.00%	0.00%	0.71/0.71	0.00%	0.00%	1.14/1.14	0.00%	0.00%
r250	1.60	0.33/0.33	0.00%	0.00%	0.00/0.00	0.00%	0.00%	0.00/0.00	0.00%	0.00%
quasi	2.80	1.25/1.25	0.00%	0.00%	0.75/0.75	0.00%	0.00%	0.00/0.00	0.00%	0.00%
Average	2.04	3.04/2.65	7.69%	2.23%	1.45/1.49	0.00%	1.52%	0.89/0.94	0.00%	1.62%

In particular, for parameter recovery, we report the implicit parameters accessed (i.e., the global variables) per function (the 2^{nd} column). Next, we report in the 3^{rd} column the average number of explicit parameters (R) per function identified by REFIT and the ground truth (T), as well as the average false positive (FP) and false negative (FN) compared with ground truth. Similarly, we report the number of syntactic data structure identified by REFIT, as well as the ground truth, and its FP and FN, from the 6^{th} column to the 8^{th} column; the result for the semantic data structure identification from the 9^{th} column to the 11^{th} column, respectively. Finally, we report the analysis time spent for each benchmark program in the last column.

From this table, we can observer that REFIT performs very well. Due to dynamic execution that did not touch the arguments, we mainly have FN for parameter, syntactic and semantics discovery. While it has 15.38% FP in parameter recovery for bcrypt-1.1, we examined the reason that is because of we loose the hierarchy structure if a structure is passed as a parameter, such as function `initoptions`, `readfile`. More specifically, as shown in

Figure 4.8, function `initoptions` only takes one argument. However, our analysis reveals it has 6 parameters, leading to FP for parameter discovery. In this case, it also results in 7.04% FN for `bcrypt-1.1` in syntactic discovery. For binary program `sum`, its FP is because of `long long` type (that is 64 bits), and REFIT mistakenly recovered two arguments of one `long long` type.

```
typedef struct _BCoptions {
   unsigned char remove;
   unsigned char standardout;
   unsigned char compression;
   unsigned char type;
   uLong origsize;
   unsigned char securedelete;
} BCoptions;

BCoptions initoptions(BCoptions options)
```

Figure 4.8. Passing Big Structure as Argument.

4.3.2 Evaluation with Binary Code Reuse

The goal of REFIT is to support binary code reuse. To demonstrate this feature, we extracted the binary code of the functions of interest from each of the binary programs (using the lined assembly approach proposed in BCR (Caballero et al., 2010)), and then reuse them when developing the third party software.

The statistics of reused functions are presented in Table 4.3 and 4.4. In particular, the addresses of reused functions are listed in 2^{nd} column of Table 4.3, followed by their symbol name in the 3^{rd} column. Finally, we report the extracted code size for the target function (and its user-defined callees) in the last column of Table 4.3. Similar to Table 4.2, we, in Table 4.4, report the D/S of user-defined functions called by the target function in the 2^{th} column. Next, for each of the function, we present the detailed result of parameter recovery,

syntactic discovery, and semantic discovery for each of the arguments, from the 3^{th} to 12^{th} column, respectively.

Table 4.3. Statistics of the Recovered Functions (I)

Binary Programs	Target Functions	Symbol Name	Size (Bytes)
sum	0x80494a0	bsd_sum_file	1534
	0x8049250	sysv_sum_file	1389
bcrypt-1.1	0x804a7d4	BFEncrypt	421
	0x804a9b8	BFDecrypt	666
tsort	0x8049520	search_item	266
bsdsort-1.0	0x804dca0	sradixsort	301
ncompress	0x804a00a	compress	245
	0x804a6c9	decompress	558
r250	0x8048434	r250_init	140
	0x804851d	r250	33
	0x804858f	dr250	41
quasi	0x80487ae	QuasiRandomInitialize	286
	0x804889f	QuasiRandomNumber	81
	0x804885d	QuasiRandomRelease	22

Table 4.4. Statistics of the Recovered Functions (II)

symbol Name	#Functions	Parameter Discovery				Syntactic Discovery			Semantics Discovery		
	(D/S)	Var	(R/T)	FP	FN	(R/T)	FP	FN	(R/T)	FP	FN
bsd_sum_file	8/9	5	2/2	0%	0%	1/1	0%	0%	1/1	0%	0%
sysv_sum_file	5/5	5	2/2	0%	0%	1/1	0%	0%	1/1	0%	0%
BFEncrypt	5/6	2	4/4	0%	0%	3/3	0%	0%	0/0	0%	0%
BFDecrypt	5/9	3	5/5	0%	0%	4/4	0%	0%	0/0	0%	0%
search_item	1/5	0	2/2	0%	0%	2/2	0%	0%	1/1	0%	0%
sradixsort	2/2	9	4/4	0%	0%	2/2	0%	0%	0/0	0%	0%
compress	1/4	10	2/2	0%	0%	0/0	0%	0%	2/2	0%	0%
decompress	1/4	11	2/2	0%	0%	0/0	0%	0%	2/2	0%	0%
r250_init	3/3	4	1/1	0%	0%	0/0	0%	0%	0/0	0%	0%
r250	1/1	1	0/0	0%	0%	0/0	0%	0%	0/0	0%	0%
dr250	1/1	2	0/0	0%	0%	0/0	0%	0%	0/0	0%	0%
QuasiRandomInitialize	2/2	7	2/2	0%	0%	0/0	0%	0%	0/0	0%	0%
QuasiRandomNumber	1/1	4	1/1	0%	0%	0/0	0%	0%	0/0	0%	0%
QuasiRandomRelease	1/1	2	1/1	0%	0%	0/0	0%	0%	0/0	0%	0%

We could see from Table 4.4 that there is no FP and FN in our parameter, syntactics, and semantics discovery. The recovered source code size is varied from tens lines (smallest one with 22) of code to thousands of lines of code (largest one with 2,624). Next, to verify the correctness of our interface for the extracted functions, we replaced the original functions with the newly extracted one, and recompile the programs. Recall we have the source code of these binary programs. We use the same test case in our dynamic analysis to test the

extracted code, if there is any deviation, we report we did not succeed in identifying the interface and extracting the code. The result is encouraging: the identified and extracted code can faithfully reproduce the same behavior as the original one.

In terms of the performance overhead of the recovered functions, we also compare the result of the new program (composed by replace the functions with the recovered one) with the original program with same input. We did not observe any additional performance penalty and they run as fast as original program (because we generate the inlined assembly which is almost close to the original program).

4.4 Summary

We have presented REFIT, an automatic approach to recover the function interface for binary code reuse. Unlike existing approaches, REFIT not only recovers the number of the parameters, but also recovers the non-trivial types such as pointer type. Also, REFIT improves existing dynamic type inference techniques with an efficient online approach, and contains a new invariant-assisted dynamic shape analysis for the recursive data structures. Our experimental result with a number of binary programs show that REFIT is able to precisely recover the function interface of our interest, and enable their reuse.

In summary, we makes the following contributions for REFIT.

- We present REFIT, a general function interface recovery technique that can precisely discover the function interfaces including the number of the arguments and their types.

- We present a new online dynamic type inference at binary code level, and our type inference, along with a novel invariant-assisted dynamic shape analysis, handles both primitive types and composite types including recursive types such as link-lists and trees.

- We have implemented all the techniques in our prototype system REFIT, and applied it to recover function interfaces of a number of legacy binary programs. Experimental results show that REFIT achieves high accuracy, with low false positives and false negatives for the recovered functions, enabling them to be reused.

CHAPTER 5

PEMU: PROVIDING SECURE INSTRUMENTATION[1]

In this chapter, we provide a detailed explanation of our last component, which is a DBI framework (PEMU) for safe instrumentation. Specifically, we present a technical overview in Section 5.1, detailed design (Instrumentation Engine in Section 5.2 and Introspection Engine in Section 5.3), evaluation in Section 5.4 and summary in Section 5.5.

5.1 Technical Overview

We present PEMU (Zeng et al., 2015) (inherited from both PIN and QEMU), a new PIN-API compatible DBI framework that provides a whole system instrumentation but from out-of-VM introspection perspective. There are a number of goals PEMU aims to achieve. Specifically, it aims for PIN API-compatibility because of the large amount of users and rich-APIs PIN has. For instance, it has over 450 comprehensive, well-documented, easy to use instrumentation APIs. With the PIN compatible APIs, PIN plugins can be easily ported to PEMU with no or minimal modifications. Meanwhile, it aims for out-of-VM instrumentation because of the isolation requirement from security applications such as introspection. In addition, it aims for supporting a large number of different guest OSes without modification, considering that there are so many OSes with many different versions today.

The key idea to realize PEMU is to add *an additional software layer* atop an existing VM, and make our APIs self-contained. Such a design makes our system easily portable if the underlying VM has been upgraded.

[1] ©2013 ACM. Reprinted, with permission, from Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. "PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework", In Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15), pages 147-160. http://dx.doi.org/10.1145/2731186.2731201

In addition to the engineering challenges (in support of the large number of PIN APIs), we also face a number of research challenges. One is how to bridge the semantic gap (Chen and Noble, 2001) while providing the out-of-VM instrumentation abstractions (e.g., APIs) for both process and kernel introspection, and also what those abstractions should be. The second is how to design our instrumentation engine such that it works seamlessly with the translation engine provided by the underlying VM but does not introduce large overhead. The third one is how to support the existing PIN APIs by using our framework.

In this section, we first discuss the background related to our system in §5.1.1, and then motivate our research in §5.1.2. Next, we discuss how to develop a plugin using PEMU in §5.1.3, and finally we give an architecture overview of PEMU in §5.1.4.

5.1.1 In-VM vs. Out-of-VM Instrumentation

The key technique behind any DBI is the just-in-time compilation (JIT) (Narayanasamy et al., 2006; Bellard, 2005a). Basically, all the executing instructions are translated by a JIT compiler, which provides an opportunity to interpose and instrument the binary code for analysis purposes. The entire DBI infrastructure can be considered as a VM, which could be a process level VM (e.g., PIN, VALGRIND), or a system level VM (e.g., QEMU). At a high level, a VM mediates program execution by dynamically translating blocks of native code and executing them from a code cache.

Given an analysis routine (e.g., printing the executed instruction addresses), there are two different ways of instrumenting the analysis routine with the original program code, as illustrated in Figure 5.1.

- **In-VM instrumentation**. This is the easiest way. The analysis routine is directly translated together with the original code into the same code cache. The analysis routine and the original program code share the same address space, and they are executed inside the VM either at guest "ring 3" (application layer) or "ring 0" (OS kernel

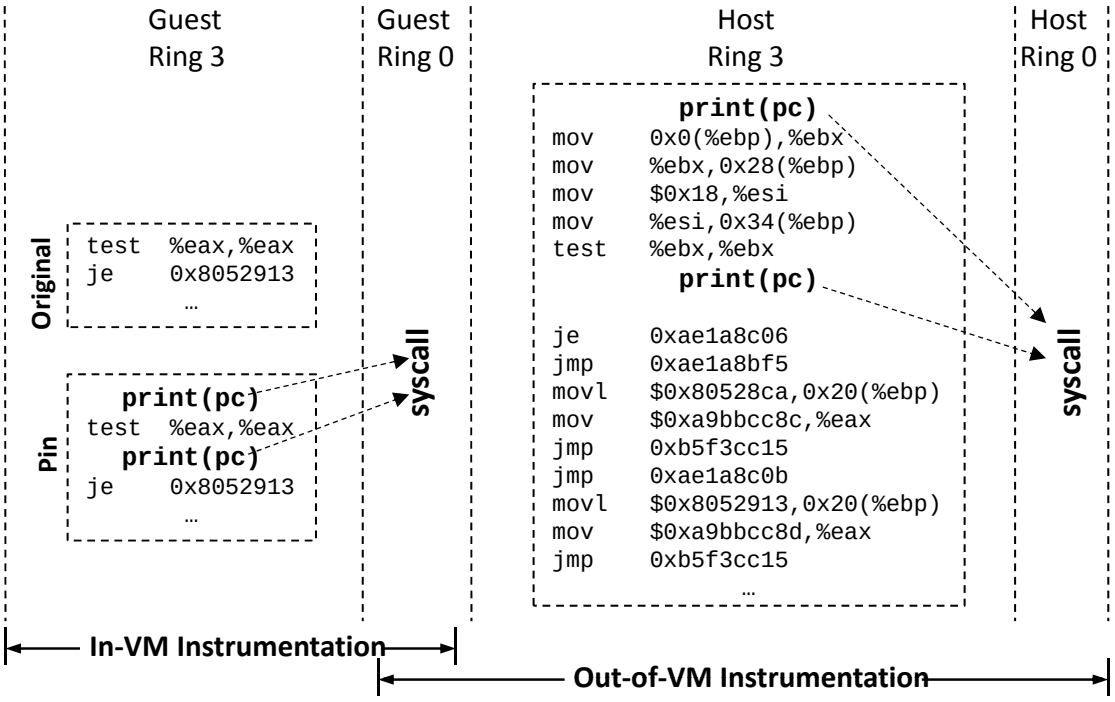

Figure 5.1. Differences Between in-VM and Out-of-VM Instrumentation.

layer). Therefore, the analysis routine can feel free to call any guest OS abstractions, and access any code or data of the instrumented process or kernel. Most DBI systems (e.g., PIN and VALGRIND) are designed in this way.

- **Out-of-VM instrumentation**. Unlike in-VM instrumentation, the analysis routine is executed outside of the original program code (mostly at the virtual machine monitor layer, e.g., at "ring 3" of a host OS), though the original code and the analysis routine can be translated into the same cache. Therefore, the analysis routine and the original code does not share the same address space any more. There is a world switch for analysis routine from host "ring 3" to access the state of the monitored process or kernel at "ring 3" or "ring 0" of guest OS. Only a handful of systems (e.g., PINOS and TEMU) support out-of-VM instrumentation. However, their introspection supports are guest OS specific. Also, the isolation provided by PINOS is not as strong as TEMU.

More specifically, while the instrumented code and instrumentation engine do not share any code in PinOS, the analysis routine and the original program code actually share the same address apace, because the analysis routine steals (Bungale and Luk, 2007) the address space from the guest OS, which makes it possible to tamper with the analysis routine when used to analyze malware.

We can notice that in-VM instrumentation and out-of-VM instrumentation share the opposite pros and cons. In-VM instrumentation occurs inside the VM, and has rich abstractions. But it executes at the same privilege level as the monitored process. Out-of-VM instrumentation occurs outside of the VM, and has less abstractions. But the analysis routine is isolated with the original program code. To develop the analysis routine, we can still use host OS abstractions, but to inspect any guest OS state, there is a need for techniques to bridge the semantic gap.

5.1.2 Objectives

While there have been significant efforts in the past 20 years to build various DBI platforms, few works focus on out-of-VM instrumentation where the analysis routine and original program code are strongly isolated. In this chapter, we would like to develop a new out-of-VM DBI with an emphasis on supporting security applications that satisfy the following constraints:

1. **Rich APIs**. Similar to Pin tool, we would like to offer rich and well-defined APIs. Since Pin is one of the most popular DBI tools, we would like to make our API compatible with Pin. This will make an open source alternative to Pin, which will be useful when there is a need to customize the Pin engine.

2. **Cross-OS**. Unlike Valgrind, which only analyzes Linux binaries, we would also like to offer support to instrument both Windows and Linux binaries using the same platform.

More importantly, since there are a large number of different OSes, we would like to make our system OS agnostic for the introspection.

3. **Strong Isolation**. Unlike existing in-VM approaches, we would like to make our analysis code execute at the hypervisor layer (can be considered "ring -1") instead of at the guest OS "ring 3" for process introspection or "ring 0" for kernel introspection.

4. **VM Introspection.** Unlike PinOS, which does not support higher level guest object introspection (Garfinkel and Rosenblum, 2003), we would like to provide APIs to retrieve the high level semantic state from the guest OS for the monitored process or guest kernel. Considering that there are too many guest OSes, we would like to design a general way to query the guest OS state.

5.1.3 An Example

Before describing the details of how we achieve these goals, we would like to first illustrate how to develop a Pemu plugin by using the provided APIs. As presented in Figure 5.2, this is a very simple plugin with the functionality of counting the number of executed instructions. Similar to many other DBIs, to develop a Pemu plugin, users need to provide two sets of procedures: *Instrumentation Routine*, which specifies where the instrumentation should occur; and *Analysis Routine*, which defines analysis activities.

One important feature of Pemu is the API compatibility with the Pin tool. As illustrated in this example, the API we used is exactly identical to those used by a Pin tool. Therefore, many legacy Pin plugins can be recompiled and executed inside Pemu, but the distinctive feature is that both the analysis routine and instrumentation routine will be executed in the host OS, instead of inside the guest OS as would be done using Pin. For instance, the `fprintf` (line 13) and `fopen` (line 21) statements will be executed at the VMM layer and call host OS `fprintf` and `fopen`, but when we use Pin they will be executed inside the guest OS.

```
1 static UINT64 icount;
2 FILE *pFile;
3 VOID docount(UINT32 c) { icount += c; }
4 VOID Trace(TRACE trace, VOID *v) {
5    for (BBL bbl = TRACE_BblHead(trace);
6       BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
7       BBL_InsertCall(bbl, IPOINT_BEFORE,
8       (AFUNPTR)docount, IARG_UINT32, BBL_NumIns(bbl),
9       IARG_END);
10    }
11 }
12 VOID Fini(INT32 code, VOID *v) {
13    fprintf(pFile, "Count %lld\n", icount);
14    fclose(pFile);
15 }
16 INT32 Usage(VOID) {
17    return 0;
18 }
19 int main(int argc, char * argv[]) {
20    if(PIN_Init(argc, argv)) return Usage();
21    pFile = fopen("pemu_count", "w");
22    TRACE_AddInstrumentFunction(Trace, 0);
23    PIN_AddFiniFunction(Fini, 0);
24    PIN_StartProgram();
25    return 0;
26 }
```

Figure 5.2. A PEMU plugin to count the number of executed instructions.

5.1.4 Architecture Overview

An overview of PEMU is presented in Figure 5.3. There are two key components inside PEMU: *instrumentation engine* (§5.2) and *introspection engine* (§5.3). They both are located inside a virtual machine monitor.

To use PEMU, end users use the PIN compatible APIs provided by our *instrumentation engine* to develop the plugins, which will be compiled and linked at the VMM layer (namely, the host OS layer). If the analysis routine requires retrieving the state of the guest OS (e.g., `pid` of the instrumented process), it uses APIs provided by our *introspection engine*. In the following two sections, we describe how we design these two engines in greater detail.

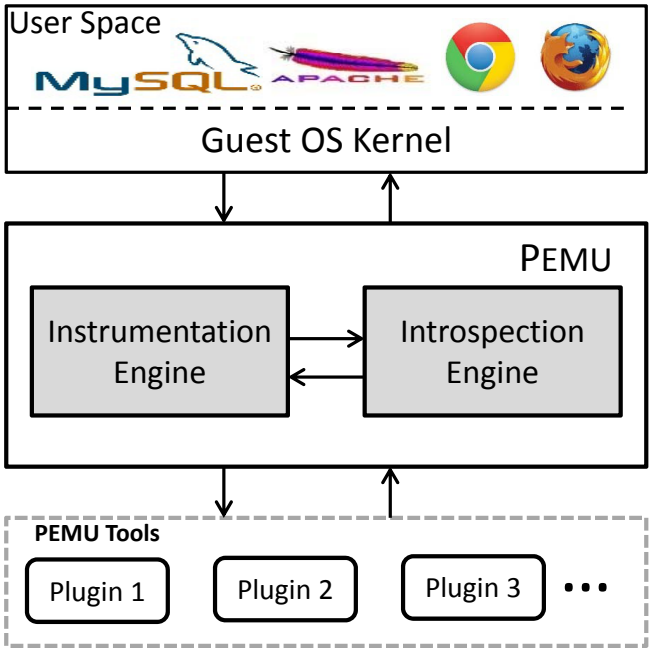

Figure 5.3. Architecture Overview of PEMU.

5.2 Instrumentation Engine

Since PEMU aims for API compatibility with PIN, we have to first examine what those PIN APIs are. We take a recently released version of PIN (version 2.13), and we find there are in total 477 APIs. The distribution of these APIs are presented in Figure 5.4.

PIN defines two sets of instrumentation: (1) *trace instrumentation* that occurs immediately before a code sequence is executed, and (2) *ahead-of-time instrumentation* that caches the instrumentation before the execution. There are three different types of *trace instrumentation* depending on the granularity:

- **Instruction Level**. The finest granularity is the instruction (INS) level instrumentation that allows for instrumenting a single instruction at a time by using the `INS_AddInstrument Function` call back. There are also many instruction insertion and inspection APIs starting with the `INS` prefix (e.g., `INS_InsertIfCall`, `INS_IsBranch`, etc.). In total there are 142 `INS` related APIs.

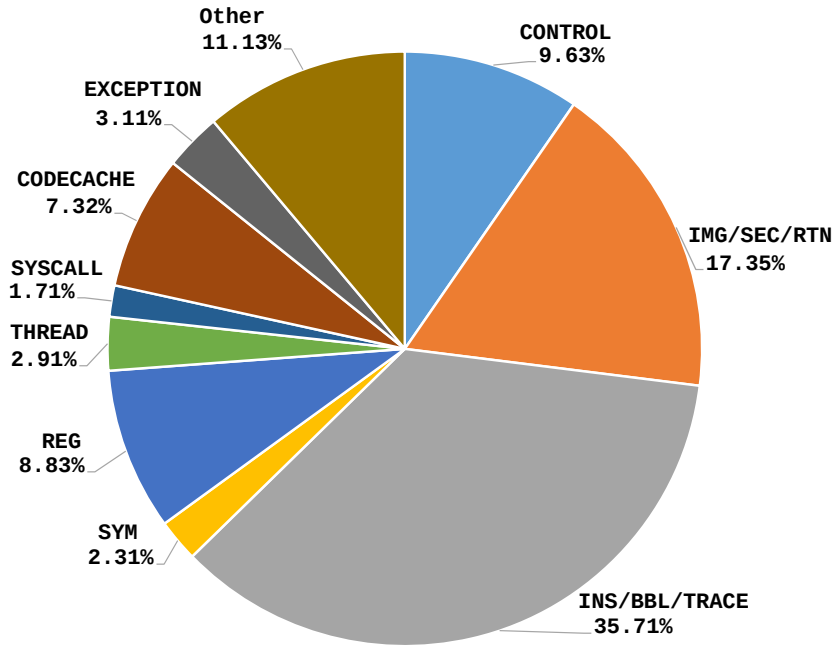

Figure 5.4. Distributions of Pin APIs.

- **Basic Block Level (BBL)**. A basic block (BB) is a single entrance, single exit sequence of instructions. Instead of one analysis call for every instruction, it is often more efficient to insert a single analysis call for a BB, thereby reducing the number of analysis calls. Pin does not offer a `BBL_AddInstrumentationFunction`, and instead developers have to instrument the TRACES (described next) and iterate through them to get the BB. There are in total 14 APIs related to BBL.

- **Trace Level**. A TRACE in Pin is defined as a sequence of instructions that begin at the target of a taken branch and end with an unconditional branch (i.e., `jmp/call/ret`). This is the set of instructions that are disassembled by a linear sweep algorithm, when giving a starting address. Therefore, a TRACE usually consists of a number of BBs. Pin provides `TRACE_AddInstrumentFunction` call back to instrument a TRACE. There are in total also 14 APIs related to TRACE. Note that Pin introduced the concept of TRACE for a trace-linking optimization (Luk et al., 2005),

which attempts to branch directly from a trace exit to the target trace without trapping to the VM. TRACE is at a higher granularity than BB and INS, and sometimes instrumenting analysis routine at TRACE granularity can improve performance. For instance, TRACE-based BBL instruction counting (as shown in Figure 5.2) is much faster than that of an INS based one.

Regarding the *ahead-of-time instrumentation*, Pin provides an image (IMG) instrumentation and routine (RTN) instrumentation. More specifically:

- **IMG instrumentation** allows a Pin-tool to inspect and instrument an entire image when it is loaded. A Pin-tool can walk the sections (SEC) of an image, the RTN of a section, and the INS of a routine. Image instrumentation relies on symbol information to determine an RTN boundary. An analysis routine can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. IMG instrumentation utilizes the `IMG_AddInstrumentFunction` API. In total, there are 27 APIs related to IMG, and 16 APIs related to SEC.

- **RTN instrumentation** allows a Pin-tool to inspect and instrument an entire routine when the image, it is contained in, is first loaded. A Pin-tool can walk the instructions of an RTN. An analysis routine can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. RTN instrumentation utilizes the `RTN_AddInstrumentFunction` API. In total, there are 39 APIs related to RTN.

Next, we discuss how to design Pemu in support of these APIs. As we base Pemu atop Qemu, we have to examine the difference between Qemu and Pin. In fact, there are substantial differences, leading to a number of new challenges while designing Pemu.

First, Qemu does not introduce any abstractions for TRACE, SEC, RTN, and IMG, and it only allows instrumentation at the INS or BB level. Therefore, we have to rebuild these

missing abstractions. Second, QEMU's disassembling is based on BB, and the size of a BB has a limited value. For example, we notice that in QEMU-1.53 a BB needs to be split if the number of generated intermediate instructions is greater than 640. But there is no such constraint in PIN.

To address these challenges, we add our own disassembler rather than using the one in QEMU. Our own disassembler aims to reconstruct the abstractions for TRACE and we thus call this component *TRACE Constructor* (§5.2.1). To insert the analysis routine into the original program code, we leverage QEMU's dynamic binary translation (DBT) engine, on top of which to design our *Code Injector* (§5.2.2). In the rest of this section, we present the detailed design of these two components.

5.2.1 TRACE Constructor

The fundamental reason to introduce our own disassembler is to build the TRACE abstraction for PIN-APIs, from which to further build many other APIs such as those related to RTN, BB, and INS. Meanwhile, to uniformly support both trace and ahead of time instrumentation, we use a cache (we call hooking point hash table) to store all the call-back points where an analysis routine is instrumented. Then whenever these instruction points are executed, they will automatically invoke the analysis routine defined by users.

Since we aim to build TRACE abstractions, which contain BB and INS, we have to disassemble per TRACE. However, QEMU disassembles an instruction at a time (per BB). Therefore, when a starting address of a TRACE is to be disassembled by QEMU, we will disassemble all of the following instructions until we encounter an unconditional branch (e.g., `jmp`, `ret`, `call`), which is the end of a TRACE. As such, we will hold an entire TRACE before QEMU disassembles each instruction inside it.

However, there are also some practical challenges. One is that the instructions that belong to a TRACE may not exist in the guest OS memory (swapped or not loaded yet). The other

```
1: Global: TPC: a set storing the starting address of a TRACE; HPHT: the global hooking point hash
   table indexed by PC and storing function pointers of the user defined analysis routine.
2: Input: PC, the current trace starting instruction address;
3: Output: a TRACE, and updated TPC and HPHT.
4: Disassemble (PC) {
5:     TRACE ← GetTRACE();
6:     BB ← GetBB();
7:     do {
8:         INST ← DisasINST(PC);
9:         InsertINST(BB, INST);
10:        if (INST_Instrument ≠ NULL) {
11:            INST_Instrument(INST, HPHT);
12:        }
13:        if (INST.type ∈ {jcc,jmp,call,ret}) {
14:            InsertBB(TRACE, BB);
15:            if (BB_Instrument ≠ NULL) {
16:                BB_Instrument(BB, HPHT);
17:            }
18:            BB ← GetBB();
19:            TPC ← TPC∪ GetTargetPC(INST) ;
20:        }
21:        PC ← PC + INST.InstLen();
22:    } while(INST.type ∉ {jmp, call, ret});
23:    if (TRACE_Instrument ≠ NULL) {
24:        TRACE_Instrument(TRACE, HPHT);
25:    }
26: }
```

Algorithm 2: *TRACE Construction*

is the instructions being disassembled are not currently being translated by QEMU-DBT, which is the underlying component for our Code Injector. Consequently, we cannot insert our analysis routines into the guest code while perform our disassembling.

To solve the first challenge, we use a proactive page fault injection approach that is triggered from the hypervisor layer and let the guest OS map the missing pages. For the second challenge, we use a global hooking point hash-table (HPHT) to cache the instruction point that will have analysis routine inserted. Later, when QEMU generates the translated code, our *Code Injector* will query this hash table to insert the analysis routine if there is any.

The Algorithm. To precisely describe how we build the TRACE abstraction and facilitate the instrumentation process, we use Algorithm 2 to show its details. For each PC that is a

trace starting address, we will start disassembling the whole TRACE (line 4-26). This is the only point to invoke our own disassembler. To decide whether a given PC is a trace starting address, we query the TPC set that stores all the starting addresses of the TRACES. Note that some of the starting address is dynamically computed, especially for indirection control flow transfers.

To disassemble a TRACE, we first create a TRACE (line 5) and a BB instance (line 6), respectively. Then we disassemble and iterate each instruction inside the basic block, and add them into the corresponding BB (line 8-9). If there is any instruction level instrumentation (e.g., when `INS_InsertCall` is called in the PEMU plugin), we add the hooking point of the disassembling instruction into the HPHT (line 10-12). Next, if there is a control flow transfer instruction (line 13-20), then the current BB ends and we insert it into the current TRACE (line 14). Also, we insert a BB hooking point if there is any (line 15-17). Meanwhile, we allocate a new BB (line 18). To get a new TRACE starting address, we invoke a helper function, GetTargetPC (line 19), and we store the new starting address in our TPC. Next, we continue to get the next instruction (line 21), which can be the next instruction inside a BB or a starting address of a new BB. Until we encounter an unconditional control flow transfer instruction, we finish disassembling the current TRACE. If there is any TRACE instrumentation, we add the TRACE hooking points into HPHT (line 23-25).

Regarding the connection between TRACE, BB, and INS, we illustrate their data structures in the right hand side of Figure 5.5. Each instance of these data structures is semantically compatible with the corresponding PIN counter-part. With these data structures, PIN's instrumentation and inspection APIs can be easily implemented. For example, when `BBL_Next` is called in a plugin, we will correspondingly traverse the BB instance list to return the next BB.

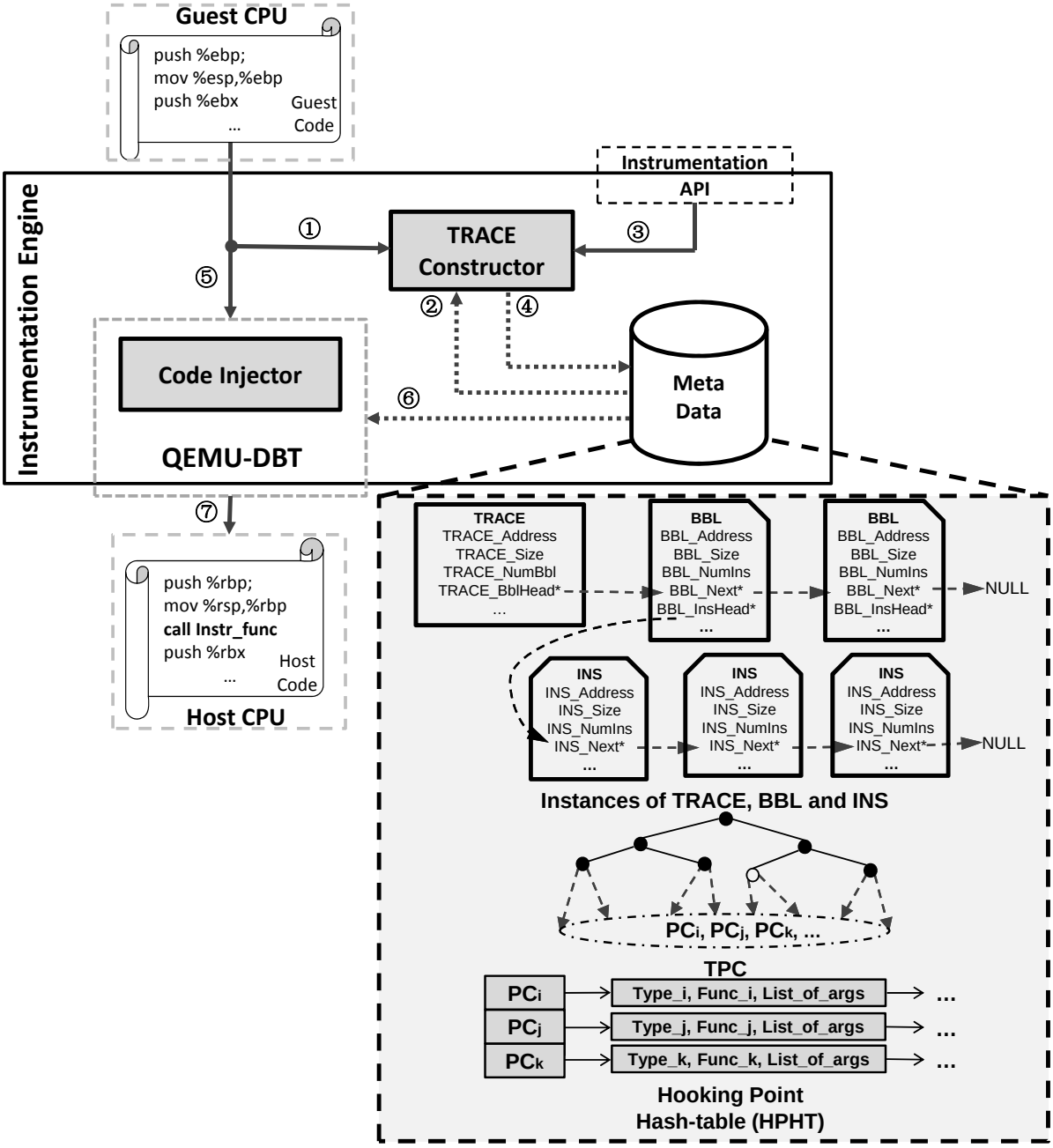

Figure 5.5. Detailed Design of Our Instrumentation Engine.

5.2.2 Code Injector

To inject the analysis routine that is specified in our HPHT, we leverage the QEMU's DBT for this goal. In particular, to translate the guest binary code into host code, QEMU uses a tiny code generator (TCG), which provides APIs to insert additional code. Having collected which instruction needs the instrumentation, our *Code Injector* will directly use the TCG API (e.g., `tcg_gen_helper`) to insert the analysis routine.

We can also notice that reconstructing TRACE abstractions as well as using the HPHT significantly alleviates the complexity of the instrumentation. With these data structures, we can uniformly achieve code injection anywhere regardless of the granularity. For instance, we can inject an analysis routine at an entry address of a BB, starting address of a TRACE, entry or exit address of an RTN, or just a particular instruction address. That is why we do not attempt to construct abstractions for RTN, SEC, and IMG. For them, we just perform ahead-of-time disassembling and extract the instruction address of interest. For instance, to hook the `malloc` routine, we just need to know the entry address of this function (which can be acquired by signature scanning in the guest memory), and then at runtime, we inject the analysis routine if there is a need for the function argument interpretation of `malloc`.

5.2.3 Putting it all together

To put it all together, we illustrate the overall execution steps of our *instrumentation engine* in Figure 5.5. For each guest instruction, our *TRACE Constructor* will take control (**Step** ①). It first checks whether the current instruction is a starting address of a TRACE by querying the metadata (**Step** ②) that stores all the observed tracing starting addresses. Note that to disassemble a new TRACE, its starting address must have been observed by QEMU, and therefore it has already been included in our TPC set (we use a red-black tree to store this set). If this is not a trace starting address, then we directly continue the execution of *Code Injector* (**Step** ⑤) to generate the final host code (**Step** ⑦). During the

code generation, our *Injector* may query the metadata, especially the HPHT data structure, to decide whether the current instruction needs an instrumentation (**Step** ⑥).

If the instruction is a TRACE starting address, then our disassembler will be invoked to disassemble the entire TRACE. During the disassembling, it will insert the corresponding instrumentation routine into the entry of the HPHT (**Step** ④), if such a routine is specified by instrumentation API in the user defined plugins (**Step** ③). When the disassembling finishes, the execution continues to Code Injector (**Step** ⑤) to generate the final host code.

5.3 Introspection Engine

Since the plugin of PEMU is executed below the guest OS, we have to design an introspection engine that supports the identification of the monitoring process/threads, as well as bridges the semantic gap when the plugins need to inspect the state of the monitored process or OS kernels.

5.3.1 Identification of Monitored Process/Threads

The instrumentation APIs and the execution of the analysis routine need to be executed when the monitored process is executing. In PIN, all of them are executed in the same address space as the monitored process. However, in PEMU, all of them are executed below the guest OS. Therefore, we have to precisely identify the target process or threads.

Given a running OS, there are a number of ways to differentiate and retrieve the process or thread execution context from a hypervisor layer. One intuitive approach is to traverse kernel data structure to locate the process name, but such an approach is OS-gnostic. Other approaches include using the value of page global directory (PGD) to differentiate each process (as shown in (Jones et al., 2006, 2008)), or using PGD and the masked value of the kernel stack pointer (as shown in (Fu and Lin, 2013)).

In PEMU, we adopted the PGD and kernel stack pointer approach. However, we still need to extend it to capture the beginning of the process/thread execution because our instrumentation happens right before process execution. To this end, we propose to capture the data life time of PGD to identify the new process. This is based on the observation that the guest OS must allocate a new (unused) PGD when creating a new process. In x86, PGD is stored in control register CR3. Therefore, if we keep tracking the use of CR3, we can detect a new process.

More specifically, starting from the execution of the guest OS, we maintain a list of the used CR3 values. When a new value is used to update the CR3 (by monitoring `mov` instructions with the destination register `cr3`), we detect that a new process is created. However, we also need to capture when the process exits, because a dead process's CR3 value can be reused for new process. Therefore, we also monitor the execution of `exit` syscall, and the CR3 value used in this syscall will be removed from the CR3 list such that we can detect a new process when this value is used again.

Note that all threads share the same address space. Therefore they will have the same CR3. To differentiate threads, we use the masked value of kernel stack pointer, because each thread will have a corresponding kernel stack that keeps the return addresses and local variables of the functions executed in a syscall trapped from the thread.

5.3.2 Addressing the Semantic Gap Challenge

Once we have detected the newly created process/thread, our instrumentation will be performed on the monitored process/thread if the instrumentation is for it. Nearly all of our instrumentation APIs are self-contained, and many of them use the abstractions provided by the host OS. Therefore, for most of the instrumentation and analysis routines, there is no semantic gap. For instance, the analysis routine can call `fprintf` in the host OS to print the analysis result.

Unfortunately, for analysis routines that inspect the running process or kernel states, we have to reconstruct their abstractions (namely bridging the semantic gap). For instance, we cannot call the `getpid` syscall at the VMM layer, because the return value of this syscall will be the `pid` of the VMM. Instead, we need to retrieve the `pid` of the monitored process executed inside the VMM.

In the past decade, many approaches have been proposed to address the semantic gap challenge. These approaches include leveraging the kernel debugging information, as shown in the pioneer work Livewire (Garfinkel and Rosenblum, 2003); analyzing and customizing kernel source code (e.g., (Petroni and Hicks, 2007; Hofmann et al., 2011)); manually creating the routines to traverse kernel objects based on kernel data structure knowledge (e.g., (Jiang et al., 2007; Payne et al., 2007)); or using a dual-VM based binary code reuse approaches (Dolan-Gavitt et al., 2011; Fu and Lin, 2012, 2013; Fu et al., 2014). Some of these approaches (e.g., (Jiang et al., 2007; Hofmann et al., 2011; Fu and Lin, 2012)) have a strong semantic gap (Jain et al., 2014), which does not trust any guest OS code; Some of them (e.g., (Fu et al., 2014)) have a weak semantic gap, which trusts guest kernel code, but not application code.

To make PEMU more practical, we adopt the approach proposed in HyperShell (Fu et al., 2014). Though it is a weak semantic gap approach, it is guest OS agnostic. More specifically, by taking this approach, we will forward the syscall execution into the guest OS if the syscall needs to inspect or retrieve the guest OS state. Regarding which syscall needs such forwarding, we let the PEMU plugin developers decide but we provide the corresponding APIs for them. For instance, if a plugin needs to retrieve the instrumented process ID, the plugin developers will invoke `PEMU_getpid`. If a plugin needs to open a file in the guest OS, it will use `PEMU_open`, and this file will be closed by `PEMU_close`. In other words, we provide a set of wrapper functions with `PEMU` prefix for state inspection and file system related `glibc`-APIs. These APIs work as usual except that we have to detour the control flow of the entry point and exit point of these syscalls, such that the corresponding

syscall execution can be forwarded to the guest OS. In total, there are 28 state inspection syscalls (including `getpid,gettimeofday` etc.), and 15 file system related syscalls (including `open,fstat,lseek`,etc.), which are forwarded to the guest OS if the plugin uses the `PEMU` prefix syscalls.

Though PEMU offers a weak semantic gap, we would like to note that for all other syscalls involved in the analysis routine, we offer a strong semantic gap. This is because we will not execute any guest code, will not traverse any guest kernel data structures, and the execution of the syscall will be directly executed on the host OS. If there is a strong security need, only the results for syscalls prefixed with `PEMU` cannot be trusted. In other words, a plugin developer is aware of this and can hence quantify the trustworthiness of her analysis routine.

Execution of a Forwarded Guest Syscall.. To illustrate how a forwarded syscall really works, we present its detailed execution steps in Figure 5.6. In general, there are three parts of code involved in an introspection process: (1) original program code, (2) the analysis routine, and (3) the modified `PEMU_glibc`.

Suppose the control flow is transferred to an analysis routine (**Step ❶**), which needs to open a file inside the guest VMby calling `PEMU_open`. Then, `PEMU_open` goes to the real `open` in `PEMU_glibc` (**Step ❷**). Next, it invokes the `syscall` function (**Step ❸**) where real `sys_open` is triggered. PEMU intercepts `syscall` so that it will not trap to host OS kernel. To forward the syscall execution to guest OS, it first needs to save the register context and set up the arguments (**Step ❹**). If the argument is a pointer, we cannot directly pass that pointer to the guest VM because the guest OS can only access memory in its address space. To allow legal memory access inside the guest OS, we inject a `sys_mmap` to allocate a piece of memory and copy the argument content to the allocated memory (here it is the file name in this case). Next, it waits until the instrumented process executes in user space, and then it forces the execution of the syscall entry (**Step ❺**). The control flow goes back to the original program and a forwarded syscall gets executed. Finally, right after the execution of the syscall exit, PEMU copies the result and restores the register context (**Step ❻**).

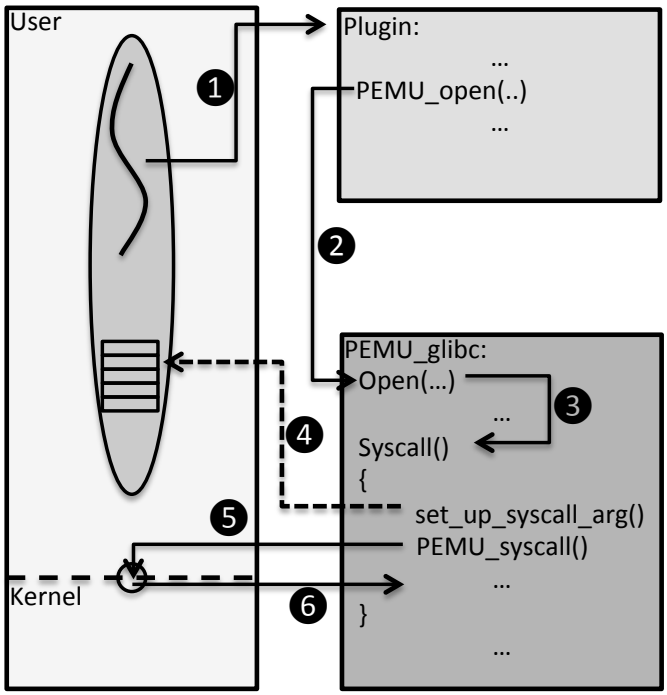

Figure 5.6. Detailed Steps For An Execution Forwarded Guest Syscall.

5.4 Evaluation

We have implemented a proof-of-concept prototype of PEMU atop `qemu-1.5.3`. We use `XED` library to build our own disassembler. Meanwhile, we have implemented over one hundred PIN compatible instrumentation APIs for INS, RTN, BB, and TRACE, as well as 43 guest OS state inspection and file system related APIs. To implement the rest of the APIs is an engineering challenge, and we leave it for future work.

In this section, we present our evaluation result. We first test the compatibility of PEMU with PIN in §5.4.1. Then in §5.4.2 we evaluate the performance of PEMU using an instruction count plugin (shown in Figure 5.2) with the `SPEC CPU2006` benchmark. Next, we evaluate the memory cost of PEMU in §5.4.3. Our host environment runs `Ubuntu 12.04` with 32-bit Linux kernel `3.0.0-31-generic-pae`, on Intel Core i7 CPU with 8G memory. Our guest OS is a 32-bit `Ubuntu 11.04` (Linux kernel `2.6.38-8-generic`) with 512M memory.

5.4.1 Compatibility Testing With PIN Plugins

To test how compatible PEMU is with PIN, we download the most recent released PIN tool, and use the plugins in `SimpleExamples` directory for this test. In total, there are 23 plugins. We recompile these plugins with PEMU's header files and library files. As shown in Table 5.1, a pleasant surprise is that 21 of them can be executed without any problem, considering that so far we only implemented over one hundred PIN APIs.

Table 5.1. Compatibility Testing with Existing PIN Plugins.

Plugin	Description	Supported
calltrace.so	Call trace tracing	✓
extmix.so	Instruction extension mix profile	✓
inscount2_vregs.so	Counting executing instructions	✓
pinatrace.so	Memory address tracing	✓
xed-cache.so	Decode cache profile	✓
catmix.so	Instruction category mix profile	✓
fence.so	Runtime text modification guard	✓
jumpmix.so	Jmp/branch/call profiling	✓
regmix.so	Register usage mix profile	✓
xed-print.so	XED usage testing	✓
coco.so	Code coverage analyzer	✓
icount.so	Counting executing instructions	✓
ldstmix.so	Register/memory operand profiler	✓
topopcode.so	Opcode mix profiler	✓
xed-use.so	XED interface usage testing	✓
dcache.so	Data cache simulation	✗
ilenmix.so	Instruction length mix profiler	✓
malloctrace.so	Tracing calls to malloc	✓
toprtn.so	Hostest routines profiling	✓
edgcnt.so	Control flow edge profiler	✓
inscount2_mt.so	Counting executing instructions	✓
opcodemix.so	Opcode mix profiler	✗
trace.so	Compressed instruction tracer	✓

More specifically, we notice that most of these test plugins are mainly used to test the tracing of instructions (including opcode and operand), control flow transfers (branching, call, ret, etc.), memory access, and library calls. Since these are the basic functionalities for a DBI tool, the current implementation of PEMU fortunately supports all of them.

As shown in Table 5.1, we have two failures `dcache.so` and `opcodemix.so`. The main reason is that our current implementation does not support APIs for `CODECACHE` and `CONTROLLER`.

Note that `CODECACHE` allows developers to inspect the PIN code cache and/or alter the code cache replacement policy, and `CONTROLLER` is used to detect the beginning or end of an interval of the execution of a program. We leave the support of these APIs for future work.

5.4.2 Performance Evaluation

Next, we test the performance of PEMU. We perform two sets of experiments: one is to measure how slow PEMU is when compared to a vanilla-QEMU, and the other is how slow when compared to PIN. We directly use the instruction counting plugin described in Figure 5.2. This plugin increases the number of instructions in a BB for an accumulated counter before the execution of each BB. We test this plugin with the SPEC 2006 benchmark programs. Each of the benchmark programs is executed 100 times, and we use the corresponding average number in our report.

Performance Comparison with vanilla-QEMU.. In this experiment, we measure the overhead introduced by PEMU instrumentation. We compare the execution when running the benchmarks with PEMU, directly with QEMU without any instrumentation.

We report the detailed experimental result in Table 5.2. Specifically, we show the total number of instructions executed in the 2^{nd} column and also the execution time of QEMU and PEMU is reported in the 3^{rd} and 4^{th} column (namely, T_{Qemu} and T_Pemu). We notice that on average there are 17649.1 million instructions traced for these benchmarks, and the average slowdown over QEMU is about 4.33X, which we believe it is reasonable for practical use. This overhead includes our TRACE Constructor, Code Injector, as well as runtime overhead of the analysis routine.

Performance Comparison with PIN.. In the second experiment, we compare PEMU against PIN using the same plugin with the same benchmark. The execution time of running in PIN is presented in the 6^{th} column, and the comparison between PEMU and PIN is presented in the last column.

Table 5.2. Performance compared with vanilla-QEMU and PIN.

Program	**#Inst (M)**	T_{Qemu} (s)	T_{Pemu} (s)	T_{Pemu} / T_{Qemu}	T_{Pin} (s)	T_{Qemu} / T_{Pin}	T_{Pemu} / T_{Pin}
401.bzip2	11500.27	24.55	81.15	3.31	11.17	2.20	7.26
403.gcc	4940.36	18.35	169.21	9.22	13.56	1.35	12.48
410.bwaves	29360.09	419.99	1336.57	3.18	7.44	56.45	179.65
416.gamess	2121.15	23.19	84.99	3.66	3.35	6.92	25.37
429.mcf	3562.67	23.91	70.58	2.95	3.55	6.74	19.88
433.milc	39509.49	779.07	2570.44	3.30	9.28	83.95	276.99
435.gromacs	4907.53	106.28	334.74	3.15	3.43	30.99	97.59
436.cactusADM	9730.11	304.89	1019.89	3.35	4.42	68.98	230.74
437.leslie3d	55857.54	900.01	3009.06	3.34	15.38	58.52	195.65
444.namd	74037.63	1523.78	5037.00	3.31	16.22	93.94	310.54
445.gobmk	314.88	2.43	4.17	1.72	1.71	1.42	2.44
450.soplex	63.67	1.49	2.22	1.49	1.80	0.83	1.23
453.povray	2987.41	36.16	193.17	5.34	3.52	10.27	54.88
454.calculix	187.33	2.53	6.61	2.61	2.46	1.03	2.69
456.hmmer	17862.2	46.43	260.56	5.61	6.95	6.68	37.49
458.sjeng	15514.49	48.40	432.79	8.94	14.38	3.37	30.10
462.libquantum	408.63	0.77	2.01	2.61	0.62	1.24	3.24
464.h264ref	98144.32	392.21	2751.31	7.01	34.01	11.53	80.90
465.tonto	3571.85	48.23	195.16	4.05	5.44	8.87	35.88
470.lbm	7744.81	161.22	692.51	4.30	2.92	55.21	237.16
471.omnetpp	2209.23	16.24	136.63	8.41	3.19	5.09	42.83
473.astar	26645.10	102.95	734.52	7.13	13.67	7.53	53.73
482.sphinx3	6198.21	77.95	322.26	4.13	4.94	15.78	65.23
999.specrand	6198.21	1.42	2.46	1.73	0.92	1.54	2.67
Avg.	17649.05	210.94	810.42	4.33	7.68	22.52	83.61

We notice that the average slowdown between PEMU and PIN is over 83.61X. The main reason is that PIN is running natively while PEMU (based on QEMU) needs extra translation. The largest slowdown comes from `444.namd` which is above 310X. However, we note that when running this program in vanilla-QEMU, it will have close to 100X slowdown. We carefully examine the reason and find the root cause due to the use of large amount of floating point instructions which needs time-consuming emulation inside QEMU.

It is also interesting to note that for `450.soplex`, running in QEMU is faster than that of PIN. The main reason is this program contains more control flow instructions that will go to the middle of a TRACE, thereby breaking the TRACE. In this case, QEMU (based on BBL disassembling) will just redisassemble the basic block that has not been disassembled, but PIN (based on TRACE disassembling) will redisassemble the whole trace after a new trace is found. Meanwhile, the running time of this program is relatively short. Thus, the time is dominated by the disassembling time.

5.4.3 Memory Cost Evaluation

Since PEMU uses an ahead-of-time instrumentation that will store the hooking point to facilitate the instrumentation, we would like to measure how much memory space this hooking point table consumes. Again, we evaluate this memory cost with our instruction counting plugin against the SPEC2006 benchmark. The result is presented in Figure 5.7. We notice that the average memory cost is about 9M.

More specifically, as shown in Figure 5.7, the maximum memory cost comes from `465.tonto` (about 22M) because this program contains the largest number of `BB`, resulting in the largest hash table to store the hooking points. More interestingly, `464.h264ref` is one of the most time consuming programs but requires a relative small size of hash table. The reason is that this program contains lots of `loops` and thus certain instructions get executed repeatedly.

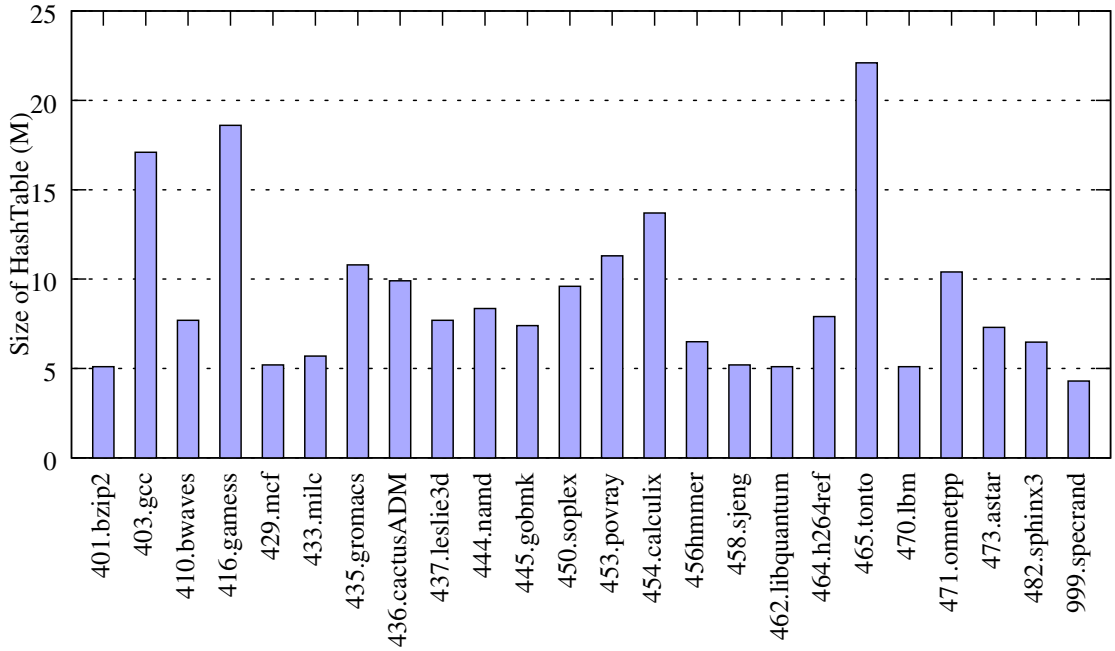

Figure 5.7. Memory Cost Comparison with SPEC2006 Benchmarks

5.5 Summary

We have presented the design, implementation, and evaluation of PEMU, a new dynamic binary code instrumentation framework that allows end-users to develop out-of-VM plugins for various program analyses. One distinctive feature of PEMU is its PIN-API compatibility. Therefore, many of the PIN-tools can be recompiled and executed within our framework. Unlike other similar systems, it is guest-OS agnostic, and can execute many different guest OSes with different versions.

In particular, we implemented PEMU atop QEMU, and our experimental results with SPEC 2006 benchmarks show that PEMU has reasonable performance overhead compared to original QEMU and it will be useful for quickly developing PIN-style plugins atop PEMU or directly recompiling the existing PIN-plugins, for both instruction inspection and higher level semantic introspection.

In summary, we makes the following contributions for PEMU.

- We devise an additional software layer atop an existing binary code translation based VM with a set of standard APIs for both user level and kernel level DBI. This additional layer hides the low level VM details and contains a number of instrumentation related abstractions. With the additional layer and the abstractions, we present PEMU, a new DBI framework that enables end-users to develop instrumentation tools using many of the existing PIN APIs.

- We have implemented PEMU and tested with SPEC 2006 benchmark. Our experimental results show that PEMU introduces reasonable performance overhead.

CHAPTER 6

APPLICATION

There are many applications related to our research work. In this chapter, we demonstrate how to apply TOP, REFIT and PEMU to handle real world problems, such as malware analysis in Section 6.1 and 6.2, reverse engineering of function interfaces in Section 6.3 and 6.4 and secure system call tracing in Section 6.5.

6.1 Malware Unpacking and Identification

Many malware programs today are heavily armored with anti-analysis mechanisms to make analysis of them difficult. Binary code packing is the most common anti-reverse engineering technique. According to a recent report, 34.79% of the malwares is packed (Rodrigo Rubira Branco and Neto, 2012). An earlier research paper (Guo et al., 2008) reports that over 80% of malware is packed. Dynamic analysis has been shown a promising approach to unpacking malware (Royal et al., 2006; Martignoni et al., 2007; Kang et al., 2007; Guo et al., 2008; Sharif et al., 2008).

Unpacking. Based on dynamic analysis, TOP naturally possesses the capability of unpacking malware. To this end, we design an unpacking plugin based on TOP. As we trace the entire execution of a binary (using layers), we are able to detect the unpacking code and the real program code. The real program code is the one that is finally settled in the memory and executed. It might happen that a packer could alternatingly execute the unpacking code and the real code (though we have not seen such a case in the wild). In that case, we might not be able to identify the real code based on full lifetime tracing. Still TOP will output all the traced code, which can be further analyzed to identify the unpacking code.

To test TOP's unpacking capability, we use 10 publicly available packers often used by malware authors and by researchers of many related efforts (Royal et al., 2006; Martignoni et al., 2007; Kang et al., 2007; Guo et al., 2008; Sharif et al., 2008)). For the "testing goat" program, we use Windows `tasklist.exe`, a command console program with a binary of 77824 bytes, to generate packed samples. The samples are created using the default configuration for all the packers. To test multi-layer packing, we deliberately create two samples that are packed by two packers. Finally we have 12 samples: 10 are packed once and two are packed twice. We run all these samples without any command line option.

The evaluation results are reported in Table 6.1. The first column shows the size of packed `tasklist.exe`, and the second column shows the packer used for packing. We report the numbers of symbolized addresses, safety checks, traced functions, and lines of code, of the generated source code. The TOP unpacking plugin can successfully detect the unpacking routine and the real `tasklist.exe` code, even if the sample binary is packed by two packers. Interestingly, as shown in the last 4 columns of Table 6.1, the real `tasklist.exe` programs generated by tracing the 12 samples (under the same configuration) have identical results across the samples. Next, we recompile the generated source code of `tasklist.exe` and execute it with the same command line option of the original program. We confirm that the outputs are the same as that generated by the original `tasklist.exe`.

Identification. Having been able to unpack malware, TOP can be further applied to malware identification. As shown in Table 6.1, each packer has its own distinctive features; and the code that follows the unpacking code belongs to the real program code. As such, we can identify the prefix of the recovered code to identify packers, and eliminate the unpacking code to expose the real program code. In particular, we could perform source code diff-ing (text-based) to identify the unpacking prefix; or we could build and normalize the program control structures (e.g., CFGs) to identify the prefix.

Table 6.1. Evaluation results with 10 publicly available packers using `tasklist.exe` as a testing goat program.

Binary Size (KB)	Packer	Unpacking Code				Recovered `tasklist.exe` Code			
		#Fun	#LOC	#Symbolized	#Safety	#Fun	#LOC	#Symbolized	#Safety
34.00	UPX	1	169	34	1	144	5748	1171	271
39.50	ASPack	14	913	140	16	144	5748	1171	271
363.50	ASProtect	19	504	86	3	144	5748	1171	271
46.46	RlPack	30	734	145	22	144	5748	1171	271
34.00	MPress	9	941	126	20	144	5748	1171	271
32.16	Mew	11	616	114	0	144	5748	1171	271
34.04	XComp	9	282	84	2	144	5748	1171	271
33.66	XPack	9	262	80	2	144	5748	1171	271
29.98	WinUnPakc	17	390	49	2	144	5748	1171	271
35.50	PEcompact	6	171	31	0	144	5748	1171	271
42.50	ASPack	14	864	130	16	144	5748	1771	271
	ASPack	14	913	140	16				
39.50	ASPack	14	797	113	16	144	5748	1771	271
	XPack	9	262	80	2				

To perform malware identification, we first generate a suite of (un)packer signatures. Given an unknown binary, we run it using Top. By comparing the output with the signatures, we eliminate the unpacking code. Then, we can apply various techniques to identify/classify the real malware code (e.g., source code diff-ing and control structure comparison). In our experiment, for simplicity, we use source text diff-ing, which indicates that the recovered programs from the samples are actually the same piece of malware. This is also confirmed by the last four columns of Table 6.1.

6.2 Security Function Transplanting

Binary code reuse is meaningful for both goodware and malware. To extract the binary code of interest, a user only needs to designate the entry point and exit point, and Top will automatically translate the executed instructions into C code. This code includes all the functions called and the symbolized global data accessed.

Goodware Function Reuse. We take an MD5 hash algorithm implementation as an example. Program `md5sum` (from coreutils) is a widely used cryptographic hash tool whose implementation contains an important function `digest_file` (instruction address: 0x8049f70 in our experiment) which computes the MD5 digest of a given file.

To avoid the effort of re-implementation, a programmer can run `md5sum` binary on TOP with tracing/translation entry point being 0x8049f70 and exit point being 0x804a07e (the `ret` instruction of function `digest_file`). TOP automatically generates the source code of all the executed functions during tracing, including `fun_0x8049f70` and 11 other sub-routines, symbolizes 151 addresses, and adds 32 checks. Next, the generated function can be reused as a normal C function in developing new software. We have successfully developed a file comparison program that performs MD5 hash check for two input files by reusing function `fun_0x8049f70` and its subroutines.

Malware Function Reuse. For demonstration purpose, we target three functions – two for environment detection and one for stream cipher – in malware code and show how to reuse them. The environment detection functions are from the two samples `anti-vm_in_instruction.exe` and `hardware_bp.exe` we have tested earlier. Our purpose is to extract the functions for VM and debugger detection and reuse them in new programs. We have successfully extracted these two functions and linked them with other software components.

We have also extracted an `RC4` stream cipher function from `Worm: Win32/Sality.AU`. Since the malware is packed, we run TOP to get the trace of the whole program, from which we identify the `RC4` cipher function. Our manual analysis reveals that the entry address of the `RC4` function is 0x401212 and the exit address is 0x402723. We then run the malware on TOP again, symbolizing 86 memory addresses and adding 24 checks for this function and its 5 callees. Now we can write a new program to reuse the cipher function and confirm the correctness of its functionality.

6.3 Interface recovery of user-defined functions

Next, we show how REFIT recovers the interfaces of user-defined function. In our recovered code, we use the names described in Figure 4.6 for primitive types. For aggregated type, we name the structure in the form of T_*pc*_*num* where *pc* is the starting address of a function

in which the aggregate type first appears, *num* is a parameter position in argument list in case that more than one aggregate type are detected in one function interface. For target functions, we use func_*pc* as the generated function name, where *pc* is the starting address of those target functions.

In the following, we show how a user-defined function interface (more specifically `BFEncrypt`) is recovered by REFIT. As illustrated in the left column of Figure 6.1, besides three primitive type arguments, this function takes a pointer argument of a `struct` type `BCoptions`. REFIT recovered interface for this function is presented in the right column of Figure 6.1.

typedef struct _BCoptions { ** unsigned char remove;** ** unsigned char standardout;** ** unsigned char compression;** ** unsigned char type;** ** uLong origsize;** ** unsigned char securedelete;** **} BCoptions;**	**struct T_804a7d4_4{** ** UNKNOWN_t elem_0x0[2];** ** num8 elem_0x2;** ** UNKNOWN_t elem_0x3[1];** ** num8 elem_0x4;** ** UNKNOWN_t elem_0x5;** **};**
uLong BFEncrypt ** (char **input,** ** char *key,** ** uLong sz,** ** BCoptions *options);**	**uint32_t func_0x804a7d4** **(num8 **arg_1,** ** num8 *arg_2,** ** uint32 arg_3,** ** struct T_804a7d4_4 *arg_4);**

Figure 6.1. Interface Recovery for `BFEncrypt`

We can notice that the type `T_804a7d4_4` is recovered, which is the type `BCoptions` and is being pointed by the last parameter of `BFEncrypt`. During online analysis of `BFEncrypt`, only two member fields (*compression* and *origsize*) of this structure are accessed, resulting in the type recovery for two of the fields (i.e., `elem_0x2` and `elem_0x4`). Note that `UNKNOWN_t` (one byte) means that the type of one memory region is unknown due to non-access (that is why it has FN as shown in Table 4.2). As to the accessed fields, REFIT types `unsigned char` with `num8` since they are only used as one byte data and no evidence for further type analysis.

Regarding other parameters, `input` is typed as `num8**` since after two times of pointer dereferecing for `input`, target memory region is accessed as one byte data. Similarly, REFIT types `key` as `num8*`. The recovered result of `sz` is `uint32` which is the same size as `uLong`.

6.4 Shape recovery for recursive data structures

In this section, we show how REFIT recovers the shape pointers from benchmark program `tsort`. The target function is `search _item` which takes two arguments: a `tree`, and a string pointer, as shown in the left column in Figure 6.2. The goal of this function is to find a node that has the same string (pointed by its *str* field) as the one pointed by the parameter `str` from the argument of a binary search tree `root`.

```
struct item
{
  const char *str;
  struct item *left, *right;
  int num;
  size_t count;
  struct item *qlink;
  struct successor *top;
};
```

```
struct T_8049520_1{
  const char *elem_0x0;
  struct T_8049520_1 *elem_tree_child_1;
  struct T_8049520_1 *elem_tree_child_2;
  num32 elem_0xc;
  UNKNOWN_t elem_0x10[12];
};
```

```
struct item *
search_item
       (struct item *root,
        const char *str)
```

```
struct T_8049520_1*
func_0x8049520
      (struct T_8049520_1 *arg_1,
       char *arg_2);
```

Figure 6.2. Examples for Shape Recovery

During dynamic execution of this function, REFIT builds an instance graph for the instances belonging to the same data structure. After leaving this function, REFIT uses a DFS traversal to traverse this instance graph and classify the pointer types, starting from `left` and `right` field of the parameter `root`. Since pointer `left` and `right` in all instances are both `tree` edges, REFIT successfully classifies it as pointer with a `tree_child` type, which explains the recovered data structure has two `elem_tree_child` fields. The last three elements of the structure are not typed since all of them are not used in our online analysis.

Also, notice this time, we recovered the second argument as `char*` instead of `num8` because we observed string operations (`strcmp` in particular) during the dynamic analysis.

6.5 Syscall Tracing for Anti-Analysis software

We have demonstrated using PEMU to analyze Linux binaries. In fact, our system is cross-OS, which is one of our design goals. To test this, we apply PEMU to analyze Windows binaries as we have evaluated with Linux binaries. In particular, we use a number of anti-PIN binaries during this test.

First, we test how PEMU would analyze the software protected by `tElock` and `safengine shielden`, which are two widely used tools to build anti-analysis software. We apply these protectors to the `hostname` binaries in a Win-XP SP3 machine, with anti-debugging and anti-instrumentation enabled, and produce two anti-analysis `hostname` binaries. We then use PIN and PEMU to analyze the packed `hostname`.

```
1 FILE *trace;
2 VOID SysBefore(ADDRINT ip, ADDRINT num) {
3    fprintf(trace,"0x%lx: %ld\n",
4           (unsigned long)ip, (long)num);
5 }
6 VOID SyscallEntry(THREADID threadIndex,
7    CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v) {
8    SysBefore(PIN_GetContextReg(ctxt, REG_INST_PTR),
9              PIN_GetSyscallNumber(ctxt, std));
10 }
11 VOID Fini(INT32 code, VOID *v) {
12    printf("program exit()\n");
13 }
14 INT32 Usage(VOID){
15    return 0;
16 }
17 int main(int argc, char * argv[]){
18    if(PIN_Init(argc, argv)) return Usage();
19    trace = fopen("strace.out", "w");
20    PIN_AddSyscallEntryFunction(SyscallEntry, 0);
21    PIN_AddFiniFunction(Fini, 0);
22    PIN_StartProgram();
23    return 0;
24 }
```

Figure 6.3. A cross-OS PEMU plugin to trace the syscall.

More specifically, we developed a simple `strace` (as shown in Figure 6.3) plugin to trace the syscall executed by the `hostname` binary. This plugin will print the syscall number at syscall entry point, and the return value at syscall exit point. We compiled it into a PIN plugin and PEMU plugin with the same source code. PIN failed on these two tests. Both packed programs detected the presence of PIN and exited at early stages. In contrast, `hostname` ran successfully on PEMU and displayed the host name.

In our other case study, we used `eXait` (Francisco Falcn, 2012), a benchmark-like tool to test anti-instrumentation techniques. `eXait` has a plugin architecture, and each technique is implemented as a separated DLL. There are 21 plugins in total. Again we run PIN with `strace` plugin to instrument `eXait` and the loaded DLLs. We found that 17 anti-instrumentation techniques detect the presence of PIN. But none of them detect the presence of PEMU.

Through these case studies, we show there is a need for out-of-VM PIN alternatives. Also, even though future malware will be able to detect the presence of PEMU, we should be able to add countermeasures against them, given that the source code of PEMU is open.

In addition, PEMU can be used to build many out-of-VM introspection tools. In the past several years, we have been using its base internally to build introspection tools such as VMST (Fu and Lin, 2012), and EXTERIOR (Fu and Lin, 2013). We believe there will be more use cases of PEMU in this regard.

CHAPTER 7

LIMITATIONS AND FUTURE WORK

In this chapter, we discuss the limitations and outline the future work of our BINARY CODE REUSE. We examine each component separately for TOP in Section 7.1, REFIT in Section 7.2 and PEMU in Section 7.3.

7.1 TOP

The main limitation of TOP is the incomplete coverage that arises from its dynamic analysis approach. The recovered code only reflects the traced behavior and rejects the behavior that is not exercised. As demonstrated in our evaluation, path coverage can be improved by advanced program testing techniques (e.g., symbolic execution (Cadar et al., 2008; Chipounov et al., 2011)). Part of our future work is to better integrate these techniques into TOP.

The current focus of TOP is its source code reconstruction and reuse capability. Hence we have not attempted to make the recovered code more readable or optimized. Our future work will address the readability issue by introducing a richer set of structures (e.g., `do-while` and `for` structures for loops) and leveraging advanced decompilation techniques (e.g., Hex-Rays (Hex-Rays, 2005), Boomerage (Emmerik and Waddington, 2004), and Phoenix (Schwartz et al., 2013), which tend to achieve better readability). We also plan to optimize the recovered code (e.g., by eliminating unnecessary safety guards). Consider the `middle_instruction.exe` example in Table 3.4. We could have removed the `jmp loc_ERROR` guard because `xor eax, eax` will clear the zero flag.

TOP is currently platform and environment dependent. More particularly, it requires the same kernel and library support to compile and execute the reconstructed code. It also

relies on the presence of the same needed external resources as the original executions, such as configuration files.

TOP currently has an effective scheme to ensure control flow safety, which is the challenge we have encountered in the programs we consider. It is possible that more complicated and subtle safety conditions may arise when TOP is applied to more complex programs. Also, the offline trace merging process simply merges control flow paths, which may not be sufficient when more extremal situations are encountered (e.g., when executions are non-deterministic). We plan to further investigate these issues.

TOP may not be able to recover the source code of all binaries. For example, a malware program may involve virtualization-based obfuscation (Sharif et al., 2009). Even though TOP can generate its source code from traces, the virtualization code will be recovered as well. Moreover, TOP cannot handle hypervisor-level malware such as the red pill, due to the lack of hypervisor-level tracing capability. This limitation will be addressed by our future work.

It is well known that broken dependences caused by control flows pose an issue for taint analysis in general. However, that is *not* an issue for TOP because we use taint analysis in a very restricted context. Specifically, to handle indirect control transfers, we use it to back track from an indirect invocation to the original instruction that loads the indirect target and replace the target with a symbol to ensure that the corresponding function becomes relocatable. The propagation from the original load of the target and the invocation must be via data dependence. It is possible that the invocation is relevant to other instructions through control dependences, yet it is unnecessary to symbolize those relevant instructions. While it is possible that aggressive obfuscation may cause problems for TOP in the future, we have not encountered such a case in our experiments.

7.2 REFIT

In this section, we examine the limitations of REFIT. The first limitation comes from the coverage issues due to the nature of our dynamic analysis. That is, the recovered function interface only reflects the behavior from what we traced, which may lead to some false negatives (as shown in §4.3.1). Similar to Howard (Slowinska et al., 2011) which combines with a symbolic execution engine (S2E (Chipounov et al., 2011) and KLEE (Cadar et al., 2008) in particular) to reverse engineer the data structure, one of our future avenues is to also leverage the advances in symbolic execution to explore more program paths and increase our coverage.

Also, our dynamic analysis currently does not support tracing multi-threaded programs. In fact, this can be addressed since each thread will have a private stack, and we can actually track the address range of `esp` register to differentiate the thread. Our another future work will solve these problems.

In addition, to apply REFIT for binary code reuse, currently we assume the knowledge of the to-be-reused functions (as we did in §4.3.2). To automatically identify these functions, we plan to integrate with our earlier dual-slicing techniques (Weeratunge et al., 2010) in our future work.

Finally, while enabling the code reuse is our primary goal and we did not attempt to make the recovered code more readable (note that BCR (Caballero et al., 2010) largely used the inlined assembly), our future work will study how to make the recovered code more readable by adopting techniques from decompilers (e.g., Hex-Ray or Boomerage (Emmerik and Waddington, 2004), as they tend to have better readability).

7.3 PEMU

The current design and implementation of PEMU has a number of limitations. The first one is the incomplete support of the PIN-APIs. Due to the grand engineering challenge,

currently we are not able to support all of the PIN-APIs. Besides continuing to finish those unsupported APIs, we would also like to leverage the power from the open source community and make PEMU open source. Being an open source PIN alternative, we believe that there will be more users of PEMU, especially when there is a need to modify the instrumentation engine.

The second limitation is that we used a weak semantic gap (Jain et al., 2014) when designing the introspection API. That is, while we did not trust any instrumented application code, we did trust the guest OS kernel because we will forward the execution of state inspection related syscalls (e.g., `PEMU_getpid`) to the guest OS. A stronger semantic gap (Jain et al., 2014) will not trust the guest OS kernel at all. How to retrieve meaningful and trustworthy information from the hypervisor layer when guest OS is untrusted is still an open challenge. One of our future works will investigate how to address this problem.

The third limitation is that we have not attempted to optimize the generated instrumentation and analysis routine yet, though we have designed a number of optimized data structures (e.g., hooking point hash table) to speed up the instrumentation process. Currently, we directly leveraged the optimization from the tiny code generator (TCG) to optimize our instrumentation and analysis routine. We leave the investigation of other optimization techniques such as leveraging parallelism (e.g., (Wang et al., 2011)) for us to pursue in another future work.

CHAPTER 8

RELATED WORK

Before concluding our dissertation, we briefly introduce the related work, including decompilation, dynamic data dependency tracking, malware analysis and unpacking, binary code extraction and reuse, data structure reverse engineering, shape analysis, dynamic data dependency tracking and so on.

8.1 Decompilation

Decompilation is the process of reconstructing program source code from code in lower-level languages (e.g., assembly or machine code) (Cifuentes, 1994; Breuer and Bowen, 1994). Tools like HexRay (Hex-Rays, 2005), Boomerage (Emmerik and Waddington, 2004), and Phoenix (Schwartz et al., 2013) offer a variety of techniques to elevate low-level assembly instructions to higher-level source code. While these techniques are all based on static analysis, TOP is based on dynamic analysis, one of the first decompilation techniques to do so.

8.2 Dynamic Data Dependency Tracking

TOP leverages dynamic data dependence tracking (i.e., taint analysis) to symbolize memory addresses. Data dependence tracking has been widely applied to many security applications, such as data lifetime tracking (Chow et al., 2004), exploit detection (Crandall et al., 2006), vulnerability discovery (Miller et al., 1990; Forrester and Miller, 2000; Cadar et al., 2006; Godefroid et al., 2008; Lin et al., 2008), protocol and data structure reverse engineering (e.g., (Caballero and Song, 2007; Cui et al., 2008; Wondracek et al., 2008; Lin and Zhang,

2008; Caballero et al., 2009; Lin et al., 2010a; Slowinska et al., 2011), and malware analysis (Egele et al., 2007)). TOP does not make any new advances in taint analysis per se but demonstrates its new application – memory address symbolization.

8.3 Malware Analysis and Unpacking

Unpacking aims at uncovering the original malicious code which had been packed by a variety of (binary) transformations. Unpacking techniques exist, such as PolyUnpack (Royal et al., 2006), OmniUnpack (Martignoni et al., 2007), Renovo (Kang et al., 2007), Justin (Guo et al., 2008), and Eureka (Egele et al., 2007). There also exist a wide range of malware analysis techniques (Moser et al., 2007; Egele et al., 2007; Kolbitsch et al., 2009, 2010). TOP complements these efforts and enables useful malware analysis capabilities such as malware identification and function extraction.

8.4 Binary Code Extraction and Reuse

Our work is closely related to BCR (Caballero et al., 2010), Inspector-gadget (Kolbitsch et al., 2010), and Virtuoso (Dolan-Gavitt et al., 2011). While the discussion of these works can be found in Section 2.2, we would like to emphasize that REFIT is the first system that primarily focuses on function interface recovery with a number of new features such as precise parameter number recovery, semantic type recovery, and syntactic (including recursive) structure recovery.

8.5 Data structure reverse engineering

Discovering of data structure knowledge is important to many applications such as memory forensics (Lin et al., 2010a) and program classification (Cozzie et al., 2008). Recently, there are a number of techniques for data structure reverse engineering (Balakrishnan and Reps,

2004, 2007; Reps and Balakrishnan, 2008; Lin et al., 2010a; Slowinska et al., 2011; Lee et al., 2011). As discussed in Section 4.2.2, REFIT complements and improves the existing techniques with an online efficient dynamic type inference.

8.6 Shape analysis

There has been a wealth of prior work on static shape analysis (David R. Chase and Zadeck, 1990; Alain Deutsch INRIA Rocquencout, 1994; Fachbereich, 1999; Manevich et al., 2005; Berdine et al., 2007; Marron et al., 2009). These techniques require flow and context-sensitive analysis, which makes them expensive and necessarily conservative. They also require access to the program's source code. Our heap data structure graph construction is inspired by the work of Marron et al. (Marron et al., 2009) and related to other approaches that summarize the state of a heap snapshot (Mitchell, 2006). Also, our approach differs from existing work in that the shape analysis in our context is to discover the edge properties from the data instances graph.

8.7 Memory Forensics

Memory forensics focuses on analyzing a memory dump to search for forensic evidence. The basic technique for memory forensics has evolved from simple string matching, to more complex object traversal (e.g., (Petroni et al., 2006; Rutkowska, Rutkowska; Case et al., 2008; Movall et al., 2005; Carbone et al., 2009)) or signature-based scanning (e.g., (Dolan-Gavitt et al., 2009; Betz, Betz; Bugcheck, 2006)). It focuses on not only live data, but also on dead memory (Lin et al., 2012); not only user level data, but also kernel level data (e.g., (Rhee et al., 2011; Feng et al., 2014)). Our work shows we can perform memory forensics by reusing the code from the existing binary for evidence interpretation and rendering.

8.8 Other DBI Frameworks

Over the past 20 years, many dynamic binary instrumentation (DBI) platforms have been developed. In this section, we compare PEMU with these platforms. Note that static binary code instrumentation or rewriting systems, including the first influential link-time instrumentation system ATOM (Srivastava and Eustace, 1994), are not within our scope.

At a high level, these dynamic binary instrumentation platforms can be classified into (1) machine simulator, emulator, and virtualizer, (2) process level instrumentation framework, and (3) system wide instrumentation framework. In the following, we discuss these related works and compare PEMU with each of them. A summary of the comparison is presented in Table 8.1.

Simulator, Emulator, and Virtualizer.. The very early development of dynamic binary code instrumentation originated from machine simulation or emulation. EMBRA (Witchel and Rosenblum, 1996) is such a simulation system. It performs whole system dynamic translation for MIPS architectures. BOCHS (community, 2001) and SIMICS (Magnusson et al., 2002) are also simulators that allow the instrumentation and inspection of all the executed x86 instructions. Targeting x86 architecture, the very early versions of VMWARE (Devine et al., 1998) also use dynamic binary translation to build virtual machine monitors (VMM). Another widely used VMM or emulator is QEMU (Bellard, 2005a), which supports a large number of architectures. When used as an emulator, QEMU uses a tiny code generator to emulate a CPU through a binary translation. QEMU can also be used as a virtualizer recently, and it can cooperate with the Xen hypervisor or KVM kernel module to achieve a near native performance through running the guest code directly on host CPU. VIRTUALBOX (Watson, 2008) is a faster VMM compared to QEMU, and it employs an in-situ patching to achieve better performance.

Table 8.1. Comparison with other dynamic binary instrumentation platforms.

Platforms	Year	Emulator, Simulator, Virtualizer	Kernel Level Instrumentation	User Level Instrumentation	w/ API for instrumentation	Out-of-VM	Guest OS Agnostic	Pin API Compatible	Open Source
Embra (Witchel and Rosenblum, 1996)	1996	✓	✓	✓	✗	✓	✗	✗	✗
Vmware (Devine et al., 1998)	1998	✓	✓	✓	✗	✓	✗	✗	✗
KernInst (Tamches and Miller, 1999)	1999	✗	✓	✓	✓	✗	✓	✗	✓
DyninstAPI (Buck and Hollingsworth, 2000)	2000	✗	✓	✓	✓	✗	✓	✗	✓
Dynamo (Bala et al., 2000)	2000	✗	✓	✓	✗	✗	✓	✗	✗
Bochs (community, 2001)	2001	✓	✓	✓	✗	✓	✗	✗	✓
Simics (Magnusson et al., 2002)	2002	✓	✓	✓	✗	✓	✗	✗	✗
Valgrind (Nethercote and Seward, 2003, 2007)	2003	✗	✗	✓	✓	✗	✓	✗	✓
Strata (Narayanasamy et al., 2006)	2003	✗	✗	✓	✓	✗	✓	✗	✓
DynamoRIO (Bala et al., 2000; Bruening et al., 2012)	2004	✗	✗	✓	✓	✗	✓	✗	✓
Qemu (Bellard, 2005a)	2005	✓	✓	✓	✗	✓	✗	✗	✓
Pin (Luk et al., 2005)	2005	✗	✗	✓	✓	✗	✓	✓	✗
Nirvana (Bhansali et al., 2006)	2006	✗	✗	✓	✓	✗	✓	✗	✗
HDTrans (Sridhar et al., 2006)	2006	✗	✗	✓	✓	✗	✓	✗	✓
VirtualBox (Watson, 2008)	2007	✓	✓	✓	✗	✓	✗	✗	✓
PinOS (Bungale and Luk, 2007)	2007	✓	✓	✓	✓	✗	✗	✓	✗
Temu (Yin and Song, 2010)	2010	✓	✓	✓	✓	✓	✓	✗	✓
Dyninst (Miller and Bernat, 2011)	2010	✗	✓	✓	✓	✗	✓	✗	✓
Drk (Peter Feiner and Goel, 2012)	2013	✗	✓	✓	✓	✗	✓	✗	✓
Decaf (Henderson et al., 2014)	2014	✓	✓	✓	✓	✓	✗	✗	✓
Pemu	2015	✓	✓	✓	✓	✓	✓	✓	✓

For all these out-of-VM works, they certainly can instrument both user level and kernel level code, but they do not offer any APIs for users to build dynamic binary instrumentation applications. Also, when used to analyze guest kernels, they all tend to be kernel specific.

Process Level Instrumentation Framework.. Recognizing the importance and wide applications of DBI, many process level instrumentation frameworks (e.g., DyninstAPI (Buck and Hollingsworth, 2000), Strata (Narayanasamy et al., 2006), DynamoRIO (Bala et al., 2000), Valgrind (Nethercote and Seward, 2003), Pin (Luk et al., 2005), Nirvana (Bhansali et al., 2006), HDTrans (Sridhar et al., 2006), Dyninst (Miller and Bernat, 2011)) have been proposed. These frameworks offer APIs for developers to build plugins for various applications such as high performance simulation (Narayanasamy et al., 2006), program shepherding (Kiriansky et al., 2002), and memory error detection (Seward and Nethercote, 2005).

Among them, Valgrind (Nethercote and Seward, 2003, 2007) is a comprehensive DBI framework that offers rich APIs for dynamic binary instrumentation. It supports various architectures (e.g., x86, ARM, MIPS) due to the use of an intermediate representation (IR) that is processor-neutral and SSA-based. Similar to Valgrind, Pin (Luk et al., 2005) also works at user space, but it only supports IA-32 and x86-64 architectures. Developers can create Pin-tools using the APIs provided by Pin, and execute them atop either Windows or Linux. It is featured with 'ease of use' with rich APIs to abstract away the underlying instruction-set idiosyncrasies. Making Pin-API compatible is one of our design goals such that Pin users can easily switch to our platform, especially when there is a need to customize the underlying DBI engine. Unlike other DBI platforms, Dyninst (Miller and Bernat, 2011) can instrument at any time in the execution of a program, from static instrumentation (i.e., binary rewriting) to dynamic instrumentation (i.e., instrumenting actively while executing the code). Also, it allows users to modify or remove instrumentation at any time, with such modifications taking immediate effect.

For process level instrumentation, they are efficient. They are built atop OS, and thus are OS-agnostic. It is also easier to develop the plugins. However, the analysis routine and the original program code share the same address space. Therefore, they are all in-VM approaches, and users have to be cautious when applying them for security sensitive applications.

System Wide Instrumentation Framework.. In addition to process level instrumentation, there is also a need for OS kernel instrumentation. KernInst (Tamches and Miller, 1999), PinOS (Bungale and Luk, 2007), Temu (Yin and Song, 2010), Drk (Peter Feiner and Goel, 2012), and Decaf (Henderson et al., 2014) are such systems.

Among them, KernInst and Drk are built atop in kernel dynamic binary code translation. They basically control all kernel instruction execution, and enable comprehensive instrumentation of the OS kernel code. PinOS (Bungale and Luk, 2007) is a whole-system instrumentation extension of Pin. It takes advantage of Intel VT Technology to interpose between the subject system and hardware. PinOS has been implemented based on the Xen virtual machine monitor. Compared to Pemu, the instrumentation and analysis code of KernInst, PinOS and Drk actually share the same address space. Even though PinOS steals the memory from the guest OS, the monitored process is still able to guess and access the memory used by analysis routines. Therefore, they do not offer strong out-of-VM isolation.

Temu (Yin and Song, 2010) is a whole-system instrumentation tool built atop Qemu. A unique feature in Temu is that it offers APIs for dynamic taint analysis and in-depth program behavioral analysis. It is an out-of-VM based instrumentation, but it installed a helper kernel module inside the guest OS to report the states to the outside analysis routine. The most recent effort, Decaf, extends Temu. It does not use any in-VM kernel module anymore, but the way to bridge the semantic gap still requires the knowledge of kernel data

structures. Therefore, DECAF is a more OS-specific solution. For TEMU, it is less since it is an in-VM based approach.

CHAPTER 9

CONCLUSION

Binary code is the final representation of many programs. This dissertation shows that we can reuse the binary code by applying dynamic binary analysis. Firstly, in Chapter 3, TOP is designed to generate the reusable source code from original executables. Second, we, in Chapter 4, introduce a function prototype recovery approach (called REFIT) that can automatically uncover function interfaces for enabling function level code reuse/calling. Finally, we propose a platform, PEMU which is able to secure our dynamic binary analysis for BINARY CODE REUSE.

TOP is proposed to generate `C` source code with templates and inlined assembly, which can be recompiled, linked with other software and reused. TOP benefits from dynamic analysis, such as being obfuscation resilient and points-to analysis free. At the same time, it overcomes code coverage shortcomings with execution safety guard and offline code combination. We extensively evaluate the effectiveness with Linux and Windows binary with and without obfuscations. Our results show that TOP works as we expected.

With the help of TOP, we can obtain reusable `C` source code. However, in order to enable code reuse in function level, we present REFIT, which leverages dynamic data flow analysis to recover the return value and argument list for function interfaces. In particular, REFIT involves not only recovery of parameter numbers, but also the syntactic information, semantic type and even shape structure for each parameter. Meanwhile, REFIT features efficiency by making dynamic analysis online without relying on offline traces. Our experimental results with `10` binary programs indicate that REFIT is able to precisely recover function interfaces, and support the binary code reuse of these functions.

As we know, Binary Code Reuse has different security applications, such as malware analysis. In other words, our target programs could be malicious to either Refit or Top. In this case, we proposed our new generation DBI framework, Pemu, for safe instrumentation. With Pemu, Refit and Top are in different address spaces than the target programs. In particular, Pemu and Refit both exist underneath the guest operating system and reside in the hypervisor layer, resulting in higher security isolation than many state-of-the-art DBI techniques. Compared to existing DBI frameworks, Pemu also has other advantages, for instance being PIN-API compatible and having whole system instrumentation. With evaluation in the `SPEC2000` benchmark, Pemu achieves acceptable performance (`4.33` and `83.61` slowdown for Qemu and Pin respectively).

Overall, Refit, Top and Pemu work together to present an integrated framework for Binary Code Reuse. Meanwhile, our research introduces many interesting research problems and we conclude this dissertation by listing some of them below.

- **Function Semantics Recovery** Function semantics indicates the meaning or internal logic of functions. Even though a function interface shows how to use the function, and thus reveals the function meaning in some sense, its semantics are still limited since (1) function interfaces specify the semantics for function entry points instead of function body logic, and (2) two of the same function interfaces could have totally different function meanings (for instance, `linked_list` insertion and `deletion` function). Therefore, uncovering of function semantics would greatly benefit Binary Code Reuse.

- **Emulation-based Obfuscation** In order to protect state-of-the-art reverse engineering approaches, obfuscation techniques have already evolved from simple code encryption, polymorphism and metamorphism to multilayered encryption and unpacking based on pages. To some extent, our Binary Code Reuse framework is able to handle the above code obfuscation. However, new obfuscation approaches (Oreans, 2009;

Total, 2009; Wu et al., 2010) recently became promising. Taking emulation-based obfuscation as an example, the emulation obfuscator essentially translates the binary code of a real instruction set architecture (ISA) into a virtual ISA. Since the obfuscator mainly takes *decode-dispatch emulation* instead of *write-and-execution*, deobfuscation would become very challenging due to instruction differentiation. Specifically, the dynamic trace of an obfuscated program is a mix of instructions from the obfuscator and the translated original programs. Recent works (Coogan et al., 2011; Sharif et al., 2009; Rolles, 2009) have achieved the goal of reverse engineering for such obfuscation technique. However, it is still not practical enough for Binary Code Reuse. As a result, deep investigation in this research area is important.

- **Code Coverage Problem** Thanks to "in the perfect light of run-time" for the memory and register state, dynamic binary analysis usually provides straightforward and precise solutions. Unfortunately, the code is not guaranteed to be exercised or traced completely (code coverage problem). For Refit and Top, the preciseness and effectiveness is greatly dependent on the code coverage since our framework Pemu is based on dynamic binary analysis. State-of-the-art researches (Cadar et al., 2006; Chipounov et al., 2011; Cadar et al., 2008; Godefroid et al., 2005; Majumdar and Sen, 2007) made a big step for this area but it still remains as an open challenge.

REFERENCES

Alain Deutsch INRIA Rocquencout, L. C. C. (1994). Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, Orlando, Florida, USA, pp. 230–241. ACM.

Aliasing (2012). Making a disassembler: Instruction aliasing. `http://trusted-disassembler.blogspot.com/2012/12/instruction-aliasing.html`.

Bala, V., E. Duesterwald, and S. Banerjia (2000). Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, New York, NY, USA, pp. 1–12. ACM.

Balakrishnan, G. and T. Reps (2004). Analyzing memory accesses in x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'04)*, pp. 5–23. Springer-Verlag.

Balakrishnan, G. and T. Reps (2007). Divine: Discovering variables in executables. In *Proceedings of Internation Conf. on Verification Model Checking and Abstract Interpretation (VMCAI)*, Nice, France. ACM Press.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, Bolton Landing, NY, USA, pp. 164–177. ACM.

Bellard, F. (2004). Tiny c compiler. http://bellard.org/tcc/.

Bellard, F. (2005a). Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA. USENIX Association.

Bellard, F. (2005b). QEMU: an open source processor emulator. www.qemu.org/.

Berdine, J., C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang (2007). Shape analysis for composite data structures. In *In CAV*.

Betz, C. Memparser. http://sourceforge.net/projects/memparser.

Bhansali, S., W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau (2006). Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, New York, NY, USA, pp. 154–163. ACM.

Branco, R. R. (2012, July). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *Black Hat Technical Security Conf.*, Las Vegas, Nevada.

Breuer, P. T. and J. P. Bowen (1994). Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst. 16*(5), 1613–1647.

Bruening, D., Q. Zhao, and S. Amarasinghe (2012). Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, New York, NY, USA, pp. 133–144. ACM.

Brumley, D., I. Jager, T. Avgerinos, and E. J. Schwartz (2011, July). BAP: A binary analysis platform. In *Proceedings of Computer Aided Verification (CAV)*.

Brumley, D. and J. Newsome (2006). Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science.

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)*, Alexandria, Virginia, USA, pp. 27–38. ACM.

Buck, B. and J. K. Hollingsworth (2000, November). An api for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*(4), 317–329.

Bugcheck (2006). Grepexec: Grepping executive objects from pool memory. *Uninformed Journal, Vol(4)*.

Bungale, P. P. and C.-K. Luk (2007). Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, pp. 137–147.

Burke, M. G., P. R. Carini, J.-D. Choi, and M. Hind (1995). Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, London, UK, pp. 234–250. Springer-Verlag.

Caballero, J., N. M. Johnson, S. McCamant, and D. Song (2010, February). Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA.

Caballero, J., P. Poosankam, C. Kreibich, and D. Song (2009). Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and and Communications Security (CCS'09)*, Chicago, Illinois, USA, pp. 621–634.

Caballero, J. and D. Song (2007). Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, Alexandria, Virginia, USA, pp. 317–329.

Cadar, C., D. Dunbar, and D. Engler (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA.

Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler (2006). Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Alexandria, Virginia, USA, pp. 322–335. ACM.

Carbone, M., W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang (2009). Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, Chicago, IL, USA, pp. 555–565.

Case, A., A. Cristina, L. Marziale, G. G. Richard, and V. Roussev (2008). Face: Automated digital evidence discovery and correlation. *Digital Investigation 5*(Supplement 1), S65 – S75. The Proceedings of the Eighth Annual DFRWS Conference.

Chen, P. M. and B. D. Noble (2001). When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pp. 133–.

Chipounov, V. and G. Candea (2010, April). Reverse engineering of binary device drivers with revnic. In *Proc. ACM SIGOPS EUROSYS'2010*, Paris, France.

Chipounov, V., V. Kuznetsov, and G. Candea (2011). S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, Newport Beach, California, USA, pp. 265–278.

Chow, J., B. Pfaff, K. Christopher, and M. Rosenblum (2004). Understanding data lifetime via whole-system simulation. In *Proceedings of the 13th USENIX Security Symposium*.

Cifuentes, C. (1994). Reverse Compilation Techniques.

Cifuentes, C. and K. J. Gough (1995, July). Decompilation of binary programs. *Softw. Pract. Exper. 25*(7), 811–829.

Collberg, C., C. Thomborson, and D. Low (1997, March). A taxonomy of obfuscating transformations. In *Technical Report (CS-2000-12), Department of Computer Science, The University of Auckland*, Auckland, New Zealand.

Collberg, C., C. Thomborson, and D. Low (2003, May). Static and dynamic analysis: synergy and duality. In *In Proceedings of the ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, Oregon, USA.

community, B. (2001). bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net/.

Constantin, L. (2011). Decompiled stuxnet code published online. http://news.softpedia.com/news/Anonymous-Publishes-Decompiled-Stuxnet-Code-184448.shtml.

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.

Cormen, T., C. Leiserson, R. Rivest, and C. Stein (2009). Introduction to algorithms (third edition). London, England. The MIT Press.

Cozzie, A., F. Stratton, H. Xue, and S. T. King (December, 2008). Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, San Diego, CA, pp. 231–244.

Crandall, J. R., S. F. Wu, and F. T. Chong (2006). Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim. 3*(4), 359–389.

Cui, W., M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz (2008, October). Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, Virginia, USA, pp. 391–402.

DARPA (2011). Research announcement: Binary executable transforms.

David R. Chase, M. W. and F. K. Zadeck (1990). Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, New York, USA, pp. 296–310. ACM.

de Boyne Pollard, J. (2004). The gen on function calling conventions.

Debray, S. K., R. Muth, and M. Weippert (1998). Alias analysis of executable code. In *Symposium on Principles of Programming Languages (POPL'98)*, pp. 12–24.

Deng, Z., X. Zhang, and D. Xu (2013, September). Bistro: Binary component extraction and embedding for software security applications. In *Proceedings of 18th European Symposium on Research in Computer Security (ESORICS'13)*, Egham, UK. LNCS.

Designer, S. (1997, August). "return-to-libc" attack.

Desoli, G., N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher (2002, November). Deli: A new run-time control point. In *In Proceedings of the 35th Annual Symposium on Microarchitecture (MICRO35)*, Istanbul, Turkey.

Devine, S. W., E. Bugnion, and M. Rosenblum (1998). Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture.

Dolan-Gavitt, B., T. Leek, M. Zhivich, J. Giffin, and W. Lee (2011). Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 297–312.

Dolan-Gavitt, B., A. Srivastava, P. Traynor, and J. Giffin (2009). Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, Chicago, Illinois, USA, pp. 566–577. ACM.

Egele, M., C. Kruegel, E. Kirda, H. Yin, and D. Song (2007). Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, Berkeley, CA, USA, pp. 18:1–18:14. USENIX Association.

Emmerik, M. V. and T. Waddington (2004). Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 27–36.

Fachbereich, R. W. (1999). Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

Feng, Q., A. Prakash, H. Yin, and Z. Lin (2014, December). Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, New Orleans, Louisiana.

Forrester, J. E. and B. P. Miller (2000). An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium*, Seattle, Washington, pp. 1–10. USENIX Association.

Francisco Falcn, N. R. (2012). Dynamic binary instrumentation frameworks: I know you're there spying on me. In *recon*.

Fu, Y. and Z. Lin (2012, May). Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of* 33^{rd} *IEEE Symposium on Security and Privacy*.

Fu, Y. and Z. Lin (2013, March). Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments*, Houston, TX.

Fu, Y., J. Zeng, and Z. Lin (2014, June). Hypershell: A practical hypervisor layer guest os shell for automated in-vm management. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Philadephia, PA.

Garfinkel, T. and M. Rosenblum (2003, February). A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)*.

Godefroid, P., N. Klarlund, and K. Sen (2005). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, pp. 213–223. ACM.

Godefroid, P., M. Levin, and D. Molnar (2008, February). Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA.

Guo, F., P. Ferrie, and T. cker Chiueh (2008, September). A study of the packer problem and its solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, Boston, USA.

Guo, P. J., J. H. Perkins, S. McCamant, and M. D. Ernst (2006). Dynamic inference of abstract types. In *Proceedings of the 2006 International Symposium on Software testing and analysis (ISSTA'06)*, Portland, Maine, USA, pp. 255–265. ACM.

Henderson, A., A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin (2014). Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, New York, NY, USA, pp. 248–258. ACM.

Hex-Rays (2005). Hex-rays decompiler SDK. http://www.hex-rays.com/.

Hofmann, O. S., A. M. Dunn, S. Kim, I. Roy, and E. Witchel (2011). Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, Newport Beach, California, USA, pp. 279–290.

Intel (2012). Intel-64 and ia-32 architectures software developer's manual combined volumes 3a, 3b, and 3c.

Jain, B., M. B. Baig, D. Zhang, D. E. Porter, and R. Sion (2014). Sok: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA, pp. 605–620. IEEE Computer Society.

Jiang, X., X. Wang, and D. Xu (2007). Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, Alexandria, Virginia, USA, pp. 128–138. ACM.

Johnson, N. M., J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. (2011). Differential slicing: Identifying causal execution differences for security applications. In *In Proceedings of the 2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Jones, S. T., A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau (2006). Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Boston, MA. USENIX Association.

Jones, S. T., A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau (2008). Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, Seattle, WA, USA, pp. 91–100. ACM.

Kang, M. G., P. Poosankam, and H. Yin (2007). Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, New York, NY, USA, pp. 46–53. ACM.

Kim, D., W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal (2014, June 7). Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, Hyderabad, India.

Kiriansky, V., D. Bruening, and S. P. Amarasinghe (2002). Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA, pp. 191–206. USENIX Association.

Kolbitsch, C., P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang (2009). Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, Montreal, Canada, pp. 351–366.

Kolbitsch, C., T. Holz, C. Kruegel, and E. Kirda (2010, May). Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of 2010 IEEE Security and Privacy*, Oakland, CA.

Lee, J., T. Avgerinos, and D. Brumley (2011, February). Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA.

Liang, D. and M. J. Harrold (1999). Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*, Toulouse, France, pp. 199–215. Springer-Verlag.

Lin, Z., J. Rhee, C. Wu, X. Zhang, and D. Xu (2012, February). Dimsum: Discovering semantic data of interest from un-mappable with confidence. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA.

Lin, Z., J. Rhee, X. Zhang, D. Xu, and X. Jiang (2011, February). Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA.

Lin, Z. and X. Zhang (2008, November). Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, Atlanta, GA, USA.

Lin, Z., X. Zhang, and D. Xu (2008, June). Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2008)*, Anchorage, Alaska, USA.

Lin, Z., X. Zhang, and D. Xu (2010a, February). Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA.

Lin, Z., X. Zhang, and D. Xu (2010b, June). Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2010)*, Chicago, IL, USA.

Lu, S., J. Tucek, F. Qin, and Y. Zhou (2006). Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, New York, NY, USA, pp. 37–48. ACM.

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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, pp. 190–200.

Magnusson, P. S., M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner (2002, February). Simics: A full system simulation platform. *Computer 35*(2), 50–58.

Majumdar, R. and K. Sen (2007). Hybrid concolic testing. In *In Intl. Conf. on Software Engineering*.

Manevich, R., E. Yahav, G. Ramalingam, and M. Sagiv. (Jan. 2005). Predicate abstraction and canonical abstraction for singly-linked lists. In *In VMCAI*.

Marron, M., D. Kapur, and M. Hermenegildo (June 2009). Identification of logically related heap regions. In *In ISMM*.

Martignoni, L., M. Christodorescu, and S. Jha (2007). Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, pp. 431–441.

Miecznikowski, J. and L. J. Hendren (2002). Decompiling java bytecode: Problems, traps and pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, London, UK, UK, pp. 111–127. Springer-Verlag.

Miller, B. P. and A. R. Bernat (2011, September). Anywhere, any time binary instrumentation. In *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Szeged Hungary.

Miller, B. P., L. Fredriksen, and B. So (1990). An empirical study of the reliability of UNIX utilities. In *Proceedings of the Workshop of Parallel and Distributed Debugging*, pp. 9–19,. Academic Medicine.

Mitchell, N. (July 2006). The runtime structure of object ownership. In *In ECOOP*.

Moser, A., C. Kruegel, and E. Kirda (2007). Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Washington, DC, USA, pp. 231–245. IEEE Computer Society.

Movall, P., W. Nelson, and S. Wetzstein (2005). Linux physical memory analysis. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*, Anaheim, CA, pp. 23–32. USENIX Association.

Mycroft, A. (1999). Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP'99)*, London, UK, pp. 208–223. Springer-Verlag.

Narayanasamy, S., C. Pereira, and B. Calder (2006). Recording shared memory dependencies using strata. In *ASPLOS'06: 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 229–240.

Narayanasamy, S., C. Pereira, H. Patil, R. Cohn, and B. Calder (2006). Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '06/Performance '06, New York, NY, USA, pp. 216–227. ACM.

Nergal (2001). The advanced return-into-lib(c) exploits: Pax case study. *Phrack 10(58)*.

Nethercote, N. and J. Seward (2003). Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV'03)*.

Nethercote, N. and J. Seward (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language design and Implementation (PLDI'07)*, San Diego, CA.

NETMARKETSHARE (2015, March). http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0.

Newsome, J. and D. Song (2005, February). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*, San Diego, CA.

Oreans (2009). Code virtualizer. http://www.oreans.com/codevirtualizer.php.

Payne, B. D., M. Carbone, and W. Lee (2007, December). Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*.

Pearce, D. J., P. H. Kelly, and C. Hankin (2007). Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst. 30*(1), 4.

Peter Feiner, A. D. B. and A. Goel (2012). Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*.

Petroni, N. L., Jr., A. Walters, T. Fraser, and W. A. Arbaugh (2006). Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3*(4), 197 – 210.

Petroni, Jr., N. L. and M. Hicks (2007). Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pp. 103–115.

Ramalingam, G., J. Field, and F. Tip (1999). Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'99)*, San Antonio, Texas, pp. 119–132. ACM.

Reps, T. W. and G. Balakrishnan (2008). Improved memory-access analysis for x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'08)*, pp. 16–35.

Rhee, J., Z. Lin, and D. Xu (2011, March). Characterizing kernel malware behavior with kernel data access patterns. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong.

Rodrigo Rubira Branco, G. N. B. and P. D. Neto (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and antivm technologies. Technical report, "NOSPAM" qualys.com.

Rolles, R. (2009). Unpacking virtualization obfuscators. In *In Proceeding WOOT'09 Proceedings of the 3rd USENIX conference on Offensive technologies.*

Royal, P., M. Halpin, D. Dagon, R. Edmonds, and W. Lee (2006). Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Washington, DC, USA, pp. 289–300. IEEE Computer Society.

Rutkowska, J. Klister v0.3. https://www.rootkit.com/newsread.php?newsid=51.

Saberi, A., Y. Fu, and Z. Lin (2014, February). Hybrid-bridge: Efficiently bridging the semantic-gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA.

Saltaformaggio, B., Z. Gu, X. Zhang, and D. Xu (2014). Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *Proceedings of the 23rd USENIX Security Symposium*, Berkeley, CA, USA. USENIX Association.

Schwartz, E. J., D. Brumley, and J. McCune (2010, February). Contractual anonymity. In *Proceedings of the Network and Distributed System Security Symposium.*

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*, Washington DC, USA. USENIX Association.

Sen, K., D. Marinov, and G. Agha (2005). Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, pp. 263–272. ACM.

Seward, J. and N. Nethercote (2005). Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA. USENIX Association.

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)*, Alexandria, Virginia, USA, pp. 552–561. ACM.

Sharif, M., A. Lanzi, J. Giffin, and W. Lee (2009). Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pp. 94–109.

Sharif, M., V. Yegneswaran, H. Saidi, and P. Porras (2008, October). Eureka: A framework for enabling static analysis on malware. In *Proceedings of the 13th European Symposium on Research in Computer Security*, Malaga, Spain. LNCS.

Slowinska, A., T. Stancescu, and H. Bos (2011, February). Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA.

Sridhar, S., J. S. Shapiro, E. Northup, and P. P. Bungale (2006). Hdtrans: An open source, low-level dynamic instrumentation system. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, New York, NY, USA, pp. 175–185. ACM.

Srivastava, A. and A. Eustace (1994). Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, New York, NY, USA, pp. 196–205. ACM.

Tamches, A. and B. P. Miller (1999). Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, Berkeley, CA, USA, pp. 117–130. USENIX Association.

Total, V. (2009). http://www.virustotal.com/.

Wallace, S. and K. Hazelwood (2007, March). Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Annual International Symposium on Code Generation and Optimization*, San Jose, CA, pp. 209–217.

Wang, Z., R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang (2011). Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, New York, NY, USA, pp. 213–222. ACM.

Watson, J. (2008, February). Virtualbox: Bits and bytes masquerading as machines.

Weeratunge, D., X. Zhang, W. N. Sumner, and S. Jagannathan. (2010). Analyzing concurrency bugs using dual slicing. In *In Proceedings of the 19th international symposium on Software testing and analysis*, Trento, Italy.

Witchel, E. and M. Rosenblum (1996). Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, New York, NY, USA, pp. 68–79. ACM.

Wondracek, G., P. Milani, C. Kruegel, and E. Kirda (2008, February). Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA.

Wu, Q., V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. Clark (2005). A dynamic compilation framework for controlling microprocessor energy and performance. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*.

Wu, Z., S. Gianvecchio, M. Xie, and H. Wang (2010). Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy*, Chicago, Illinois. ACM.

Yin, H. and D. Song (2010). Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

Yin, H., D. Song, M. Egele, C. Kruegel, and E. Kirda (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, New York, NY, USA, pp. 116–127. ACM.

Zeng, J., Y. Fu, and Z. Lin (2015, March). Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments*, Istanbul, Turkey.

Zeng, J., Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu (2013, November). Obfuscation-resilient binary code reuse through trace-oriented programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, Berlin, Germany.

Zhang, M., Q. Rui, N. Hasabnis, and R. Sekar (March, 2014). A platform for secure static binary instrumentation. In *Virtual Execution Environments (VEE)*.

VITA

Junyuan Zeng received his BE degree in Software Engineering from South China University of Technology in 2007, MS degree in Computer Science from NYU Polytechnic School of Engineering in 2010, and PhD degree in Computer Science from The University of Texas at Dallas in 2015. His research interests are mainly system and software security with a focus on binary code analysis, reverse engineering, and binary code instrumentation, with the goals of applying those techniques to analyze real-world systems for security purposes. After graduation, he will join the Mobile Security team at FireEye as a Staff Software Engineer.