January 2015

BINARY INSTRUMENTATION AND TRANSFORMATION FOR SOFTWARE SECURITY APPLICATIONS

Zhui Deng
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Zhui Deng

Entitled
BINARY INSTRUMENTATION AND TRANSFORMATION FOR SOFTWARE SECURITY APPLICATIONS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dongyan Xu
Chair

Xiangyu Zhang

Ninghui Li

Sonia Fahmy

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dongyan Xu and Xiangyu Zhang

Approved by: Sunil Prabhakar 07/24/2015

Head of the Departmental Graduate Program Date

BINARY INSTRUMENTATION AND TRANSFORMATION FOR SOFTWARE SECURITY APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Zhui Deng

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2015

Purdue University

West Lafayette, Indiana

To my wife *Yuetling Wong*

ACKNOWLEDGMENTS

First of all, I would like to thank my advisors, Professor Dongyan Xu and Professor Xiangyu Zhang for their guidance and support throughout my PhD study. Professor Xu taught me to how to do research, guided me to become a mature and responsible researcher and provided me the freedom and the facilities to do the research I was interested in. Professor Zhang helped me getting over many difficulties and challenges with his expertise in the area of binary analysis. He was always available for discussing ideas and sharing invaluable comments and suggestions. This dissertation would not have been possible without their support and advice.

I would also like to thank Professor Ninghui Li and Professor Sonia Fahmy for taking time off from their busy schedule to serve on my advisory committee. Their insightful feedback and comments helped me to significantly improve this dissertation.

My gratitude also goes to the current and former FRIENDS lab members whom I have interacted with during my graduate studies: Chao Wu, Fei Peng, Cheng Cheng, Zhongshu Gu and Cong Xu, to name a few. It has been a pleasure for me to work with these brilliant colleagues. I really enjoyed our discussion ranging from research to real life. Also, I want to thank my mentor Mengyao Li during my internship at Google for his help and advice on my career path.

Finally and most importantly, I would like to thank my wife Yuetling for her persistent and generous support over the years. She solely took care of our family while working on her PhD degree at the same time. Also, I must thank my daughter Iris who has brought me so much happiness ever since she was born. It was her smile that helped me through the toughest times in my graduate school career.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT viii

1 INTRODUCTION 1
1.1 Thesis Statement 4
1.2 Contributions 4
1.3 Dissertation Organization 6

2 BISTRO: BINARY COMPONENT EXTRACTION AND EMBEDDING 7
2.1 Introduction 7
2.2 Overview and Assumptions 10
2.3 Problems with Existing Techniques 12
2.4 Basic Algorithm for Binary Extraction/Stretching 15
2.5 Handling Indirect Control Transfer 17
2.5.1 Handling Indirect Calls 18
2.5.2 Handling Indirect Jumps 22
2.6 Handling Data References 23
2.7 Evaluation 24
2.7.1 Performance: Efficiency and Overhead 24
2.7.2 Case Study I: Binary-level Semantic Patching Using BISTRO 27
2.7.3 Case Study II: Malware Stitching Using BISTRO 32
2.7.4 Case Study III: Trojan-ing Kernel Drivers 34
2.8 Limitation 36
2.9 Related Work 37
2.10 Summary 39

3 SPIDER: STEALTHY BINARY INSTRUMENTATION VIA HARDWARE VIRTUALIZATION 41
3.1 Introduction 41
3.2 Related Work 42
3.3 Overview 46
3.4 Design 48
3.4.1 Splitting Code and Data View 48
3.4.2 Handling Breakpoints 50
3.4.3 Monitoring Virtual-to-Physical Mapping 51

Page

3.4.4 Handling Code Modification . . . 55
3.4.5 Data Watchpoint . . . 56
3.4.6 Handling Timing Side-Effect . . . 56
3.5 Implementation . . . 57
3.6 Evaluation . . . 59
3.6.1 Transparency . . . 60
3.6.2 Case Study I: Attack Provenance . . . 61
3.6.3 Case Study II: Stealthy Introspection . . . 64
3.6.4 Performance Overhead . . . 66
3.7 Summary . . . 67

4 IRIS: VETTING PRIVATE API ABUSE IN IOS APPLICATIONS . . . 69
4.1 Introduction . . . 69
4.2 iOS Background . . . 72
4.2.1 Function Invocations . . . 72
4.2.2 Private API . . . 74
4.2.3 iOS Runtime Security . . . 75
4.2.4 Execution of iOS Application . . . 76
4.3 Porting Valgrind to iOS . . . 79
4.4 Resolving API Call Targets . . . 80
4.4.1 Overview . . . 80
4.4.2 Resource Analysis . . . 82
4.4.3 Static Analysis . . . 84
4.4.4 Iterative Dynamic Analysis . . . 89
4.5 Evaluation . . . 91
4.5.1 Case Study: A Suspicious Advertisement Service Provider . . . 95
4.6 Limitation . . . 97
4.7 Related Work . . . 98
4.8 Summary . . . 100

5 CONCLUSIONS . . . 102
5.1 Future Work . . . 104

LIST OF REFERENCES . . . 106

VITA . . . 114

LIST OF TABLES

Table Page

2.1 Performance of Bistro 26

2.2 Results of binary semantic patching using Bistro 29

2.3 Trojan-ed device drivers 34

3.1 Transparency of Spider and other debuggers/DBI frameworks 59

4.1 Uses of private APIs detected by iRiS in iOS applications 92

4.2 Private API invocations in a utility application 95

LIST OF FIGURES

Figure		Page
1.1	Binary transformation and instrumentation frameworks	5
2.1	Overview of Bistro	10
2.2	Examples of detour using trampoline	13
2.3	Difficulties of binary component transplanting	14
2.4	Indirect call handling in binary stretching/extraction	17
2.5	Binary stretching with anchors	19
3.1	Overview of Spider	46
3.2	Monitoring guest virtual-to-physical mapping	52
3.3	Overhead of using Spider to perform instrumentation for BEEP	63
3.4	Overhead of Spider with regard to the number of breakpoint hits	66
4.1	Different forms of function invocations in iOS application	72
4.2	Event driven execution of iOS application	76
4.3	Overview of iRiS	81

ABSTRACT

Deng, Zhui Ph.D., Purdue University, August 2015. Binary Instrumentation and Transformation for Software Security Applications. Major Professors: Dongyan Xu and Xiangyu Zhang.

The capabilities of software analysis and manipulation are crucial to counter software security threats such as malware and vulnerabilities. Binary instrumentation and transformation are the essential techniques to enable software analysis and manipulation. However, existing approaches fail to meet requirements (e.g. flexibility, transparency) specific in software security applications.

In this dissertation, we design and implement binary instrumentation and transformation systems specifically for software security applications. First, we present Bistro, a static binary transformation framework that can extract/embed binary components from/into existing binaries without source code, symbolic or relocation information. We propose two algorithms to patch both direct and indirect control-flow transfer instructions when performing static binary transformation. Second, we present Spider, a dynamic binary instrumentation framework that enables efficient instruction-level instrumentation that is transparent to the instrumented binary program. In Spider, we propose a novel instrumentation primitive based on hardware virtualization called invisible breakpoint to replace traditional software breakpoint for better transparency, and design an algorithm to monitor the virtual-to-physical address mapping in hardware memory virtualization. Finally, we present iRiS, an iOS application vetting system for detecting private API uses. We propose a novel analysis of iOS applications using a combination of static analysis and dynamic binary instrumentation, and build iRiS on top of a dynamic binary instrumentation framework ported to iOS by us.

We build the prototypes of the three aforementioned systems and evaluate their performance against real-world binary programs. BISTRO is able to transform large-scale binary programs such as Adobe Reader, and incurs trivial runtime overhead (1.9% on average) and small space overhead (11% on average). SPIDER remains transparent against all state-of-the-art anti-instrumentation detections, and incurs reasonable overhead which is similar to hardware breakpoint. We also apply BISTRO and SPIDER in five scenarios to demonstrate their effectiveness in software security applications. iRiS successfully identified 149 (7%) malicious applications from 2019 applications that have passed the official application vetting process of iOS. From these malicious applications, iRiS has found the usage of a total number of 153 different private APIs including 28 security-critical APIs that access sensitive user information such as device serial number.

1 INTRODUCTION

Software industry has advanced significantly over the last decade. Large companies are building more and more complicated software to accommodate the ever-growing needs of software users. At the same time, Internet allows a large number of individual and small groups of software developers to create and distribute their software. This trend is further amplified by the booming sale of mobile devices such as smartphones and tablets in recent years; by the end of 2014, there are more than one million applications available in each of the two major mobile platforms, iOS and Android.

With the fast growth of both the complexity and the total number of software products, the security of software users becomes an increasing concern. The wide variety of software distribution channels enable malicious attackers to easily distribute malicious software or trojan-ed legitimate software with malicious logic embedded. Even legitimate software might contain vulnerabilities which could be exploited remotely by malicious attackers to control and steal information from the victim software user. Vulnerabilities are very common in software from individual or small teams of developers due to programming error and lack of proper tests. Even for software produced by large companies or organizations, vulnerabilities still exist and often cause significant impacts due to the large number of users. For example, the notorious Heartbleed vulnerability in the widely-used OpenSSL cryptographic library has affected around half a million websites certified by trusted authorities, allowing attackers to steal servers' private keys and user passwords.

To defend against the aforementioned attacks, it is necessary to be able to *analyze* and *manipulate* software. For example, software analysis allows users to detect malicious software or identify which component in a trojan-ed software contains malicious logic. Software manipulation allows users to remove the malicious logic in a trojan-

ed software to restore it to its legitimate state, or patch vulnerabilities in legitimate software to prevent potential attacks.

Software analysis and manipulation are challenging in practice. Although open-source software is becoming prevalent nowadays, many software products are still distributed in the form of *binary* executables. Software developers compiles source code written in a high-level programming language (e.g. C/C++) to target executable in binary form (e.g. x86/ARM instructions) to allow the program being directly executed on the CPU, which provides much faster execution than an interpreter or simulator. However, the high-level program structural information (e.g. functions, loops, classes, etc.) is stripped or lost during this process. Although it is possible to let the compiler keep that information in symbol files for the purpose of debugging, software developers, especially commercial off-the-shelf (COTS) software vendors usually choose not to disclose the symbol files in order to better protect their intellectual properties from being reverse-engineered. Therefore, the capabilities of binary program analysis and manipulation without source code and symbolic information is crucial in analyzing and manipulating practical software.

There are two types of binary program analysis and manipulation approaches: static and dynamic. Static binary transformation operates on the binary program itself and produces another stand-alone binary program that can execute on its own. On the other hand, dynamic binary instrumentation operates on the execution of the binary program. It usually involves an additional runtime environment to inject additional code to the instruction sequence executed at runtime. Although there are many existing binary transformation and instrumentation frameworks in both categories, they are either not designed specifically for security or have serious limitations that prevent their effective usage in software security applications.

None of the existing static binary transformation frameworks [1–10] supports binary component extraction and embedding, which is an important pair of primitives in a wide range of software security application scenarios. For example, in *semantic patching*, software from different vendors might contain the same third-party library

code. Suppose one vendor identifies a vulnerable function in this library and releases a patch for its software, whereas other vendors have not. With binary component extraction and embedding, we could extract the patched function in the patched software as a component and embed it into other vendors' software to replace the vulnerable version of the same function. Also, in *malware analysis*, sometimes the captured malware samples might be non-executable corpses due to various reasons (e.g. unsuccessful unpacking). In such case, binary component extraction allows us to extract executable portion from the corpse to analyze its behavior.

For dynamic binary instrumentation, the transparency of the instrumentation framework is highly desired in software security scenarios. For example, in *malware analysis*, many samples nowadays are equipped with anti-debugging and anti-instrumentation techniques to counter dynamic analysis. Such samples will cease their malicious behavior once they detect the dynamic binary instrumentation. Even in application scenarios which monitor the execution of legitimate programs, such as *high accuracy attack provenance* [11], transparent dynamic binary instrumentation is still necessary as many COTS software products are protected by advanced software protectors which checks for the existence of instrumentation to prevent reverse-engineering. Unfortunately, none of the existing dynamic binary instrumentation framework [12–29] is transparent to the instrumented binary program.

Binary transformation and instrumentation on mobile platforms have received increasing attention due to the tremendous popularity gained by Android and iOS, the two dominating mobile platforms. Although there are a lot of existing work [30–37] based on binary analysis and manipulation tools for Android, few work has been done on the iOS platform due to its closed-source nature. Compared with Android, there is no instruction-level dynamic binary instrumentation framework on iOS at all, so all existing work [38] are based on pure static analysis. Although only a few iOS malware samples have been reported in the past several years, recent advanced attacks [39,40] have shown that it is trivial to bypass Apple's App Review process and other approaches based on pure static analysis. Therefore, an important challenge is

to build a strong application vetting system based on dynamic binary instrumentation to guarantee the security of the iOS application users.

1.1 Thesis Statement

This dissertation address three important challenges in binary instrumentation and transformation for software security applications. More specifically, we focus on (1) enabling binary component extraction and embedding in static binary transformation; (2) providing complete transparency in dynamic binary instrumentation and (3) enabling and applying dynamic binary instrumentation on iOS mobile platform.

This dissertation demonstrates the following statements:

- It is possible to perform static transformation which supports binary component extraction and embedding on binary program without source code, symbolic or relocation information.
- It is possible to build a dynamic instrumentation framework which is completely transparent to the instrumented binary program based on the features in commodity CPUs.
- It is possible to build a dynamic binary instrumentation framework on iOS mobile platform, and an iOS application vetting system based on dynamic binary instrumentation can substantially increase the effectiveness of application vetting compared with approaches purely based on static analysis.

1.2 Contributions

Our contributions could be summarized as follows.

- As illustrated in Figure 1.1, we propose a static binary transformation framework called BISTRO[1] to extract/embed binary component from/into existing

[1]BISTRO stands for BInary STRetching Operation.

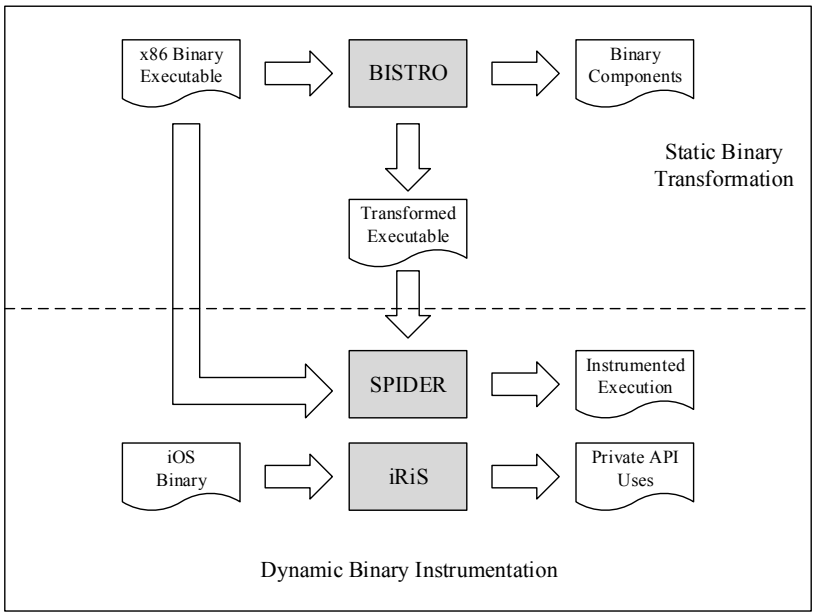


Figure 1.1.: Binary transformation and instrumentation frameworks for software security applications.

binary programs without source code, symbolic or relocation information. We propose two algorithms to patch both direct and indirect control-flow instructions in the target binary program to preserve its functional correctness.

- We propose a dynamic binary instrumentation framework called SPIDER[2] that is transparent to the instrumented program. In order to achieve transparency, we propose a novel instrumentation primitive called invisible breakpoint based on hardware virtualization to replace software breakpoint in traditional instrumentation.

- We have implemented a prototype of BISTRO for x86 Win32 PE binaries, and a prototype of SPIDER for both Windows and Linux guests on x86. We evaluated their effectiveness and efficiency in many software security applications. In particular, we use BISTRO to patch vulnerabilities, extract executable components from malware corpse and stitch malware samples for penetration tests.

[2]SPIDER is the acronym of Stealthy binary Program Instrumentation and Debugging EnviRonment.

We use SPIDER to improve instrumentation in attack provenance and capture confidential information in digital forensics.

- We have ported Valgrind [18] to enable dynamic binary instrumentation on iOS mobile platform. Based on the dynamic binary instrumentation framework, we have designed and implemented an automated iOS application vetting system called iRiS which uses a combination of static and dynamic analysis. We have applied iRiS to detect user privacy leakage due to the use of security-critical private APIs in iOS applications.

1.3 Dissertation Organization

This dissertation consists of four chapters. Following this introductory chapter, Chapter 2 presents the design and implementation of BISTRO, our static binary transformation framework. Chapter 3 presents the design and implementation of SPIDER, our dynamic binary instrumentation framework. In Chapter 2 and 3 we also evaluate the performance of our framework and demonstrate its application in several software security and malware analysis scenarios. Chapter 4 presents the design and implementation of iRiS, our binary instrumentation and analysis system on iOS mobile platform. In Chapter 5 we conclude and present our future work.

2 BISTRO: BINARY COMPONENT EXTRACTION AND EMBEDDING

2.1 Introduction

In software security and malware analysis, researchers often need to manipulate binary code – benign or malicious – without source code or symbolic information. One pair of complementary binary manipulation primitives is to (1) extract a re-usable functional component from a binary program and (2) embed a value-added functional component in an existing binary program. We call the binary manipulation primitives described above binary component *extraction* and *embedding*. These primitives are useful in a wide range of software security and malware analysis scenarios. In *security hardening of legacy binaries*, binary component embedding enables the retrofitting of legacy or close-source software with a third-party functional component that performs a value-added security function such as access control policy enforcement. In *binary semantic patching*, binary programs from different vendors may leverage the same functional component. Suppose one vendor identifies a vulnerability in such a component and releases a patched version for its own program; whereas other vendors are not aware of the vulnerability or have not patched their products. We can apply binary component extraction to carve out the patched component from a patched program and replace the vulnerable version of the same component in an un-patched program using binary component embedding. In *malware analysis*, binary component extraction and embedding supports "plug and play" of malicious functions extracted from malware captured in the wild. One can even "stitch" multiple extracted malware functions to compose a new piece of malware – a capability that may help enable *strategic defence* in cyber warfare.

Enabling binary component extraction and embedding poses significant challenges. Brute force extraction and insertion of binary functions will most likely fail. Instead,

both the extracted component and the target binary program need to be carefully transformed. For example, instructions in the target binary need to be shifted to create space for the embedded function; when a function is extracted from its origin binary, the instructions in it need to be re-positioned and re-packaged; accesses to global variables need to be re-positioned; function pointers need to be properly handled; and indirect jumps/calls need to have their target addresses recalculated. These problems are especially challenging when the binary component or the target binary program is *not relocatable*, which is often the case when dealing with legacy or malware binaries.

Despite advances in binary instrumentation and rewriting, existing techniques are inadequate to address the binary component extraction and embedding challenges. Dynamic binary instrumentation tools such as PIN [17], Valgrind [18], DynamoRIO [19] and QEMU [41] perform instrumentation only when a binary program is executed on their infrastructures. They do not generate an instrumented, standalone version of the binary for production runs. Static binary rewriting tools such as Diablo [1], Alto [2], Vulcan [3], and Atom [4] can generate instrumented, standalone binaries. However, they require symbolic information or that the binaries be generated by special compilers.

More lightweight techniques exist that do not require symbolic information or special compilers [5–10]. Among these techniques, some create *trampolines* at the end of the target binary program in which instrumentation is placed and then use control flow *detours* to access the trampolines [5–7]. The others duplicate the body of the target binary program in its virtual memory space and only the replica is instrumented. The original binary body is retained in its original position to provide a kind of control flow forwarding mechanism [8–10]. However, none of these techniques supports extraction of binary component or implanting an extracted component to another binary. Many of them cause substantial space/performance overhead. To the best of our knowledge, none of them has been successfully applied to large-scale

Windows applications or kernel code. A more detailed comparison is presented in Section 2.3.

Recently, researchers proposed approaches that focus on identification, extraction and reuse of components from binaries. Inspector Gadget [42] performs dynamic slicing to identify and extract components from malware. The extracted component might have incomplete code path coverage due to the limitation of dynamic analysis. BCR [43] adopts a combination of static and dynamic approach to extract a function from a binary. However, it uses labels to represent jump/call targets, thus does not preserve the semantic of indirect jumps/calls. ROC [44] uses dynamic slicing to identify reusable functional components in a binary but does not extract them. None of them supports reusing extracted components to enhance legacy binaries. Moreover, they could not extract components from non-executable binaries (e.g., malware corpse) due to the use of dynamic analysis.

In this chapter, we present Bistro, a systematic approach to binary functional component extraction and embedding. Bistro automatically performs the following: (1) extracting a functional component, with its instructions and data section entries non-contiguously located in the virtual address space, from an original binary program and (2) embedding a binary component of any size at any user-specified location in a target binary program, without requiring symbolic information, relocation information, or compiler support. For both extraction and embedding, Bistro preserves the functionalities of the target binary program and the extracted component by accurately patching them – using the same approach and technique. Bistro performs extraction and embedding operations efficiently and the "stretched" target binary program after embedding only incurs small time and space overhead.

We have developed a prototype of Bistro as a IDA-Pro [13] plugin. We have conducted extensive evaluation and case studies using real-world Windows-based applications (including large-scale software such as Firefox and Adobe Reader), kernel-level device drivers, and malware. Our evaluation (Section 2.7) indicates Bistro's efficiency and precision in patching the extracted components and target binary pro-

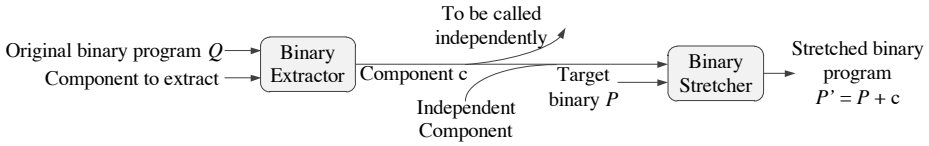


Figure 2.1.: Overview of BISTRO.

grams. Moreover, the stretched target binary program incurs small performance overhead (1.9% on average) and space overhead (10.9% on average). We have applied BISTRO to the following usage cases: (1) We carve out patched components from a binary and use them to replace their un-patched versions in other application binaries, achieving binary semantic patching (Section 2.7.2); (2) We stitch malicious functions from an un-executable Conficker worm [45] sample and compose a new, executable malware (Section 2.7.3); and (3) We demonstrate the realistic threat of *trojan-ed* device drivers with malicious rootkit functions embedded in benign driver – using real-world drivers and rootkits (Section 2.7.4).

2.2 Overview and Assumptions

An overview of BISTRO is shown in Figure 4.3. BISTRO has two key components: *binary extractor* and *binary stretcher*.

- The binary extractor is responsible for extracting a designated functional component c from an original binary program Q. c includes both the code and data of the functional component. The extractor does so by removing the unwanted code and data from Q and then collapsing the remaining data and code into a re-usable component c that occupies a contiguous virtual address region. More importantly, the instructions in c are properly patched for repositioning. We note that c can either be called as a library function or be embedded directly in another binary program.

- The binary stretcher is responsible for stretching the target binary program P to make "room" (holes in its address space) to embed a function component. As shown in Figure 4.3, the stretcher takes the target binary P and the to-be-embedded component c as input; stretches P, and patches the code in P to allow the embedding of c. The output of the stretcher is a "stretched" binary program $P' = P + c$ that is ready for execution.

Summary of Enabling Techniques. Both the binary extractor and stretcher are based on the same binary stretching algorithm (Section 2.4). The overarching idea is to shift instructions for creating space (by stretcher) or squeezing out unwanted space (by extractor). The algorithm focuses on patching the control transfer and global data reference instructions by precisely computing the offsets they need to be adjusted. For instance, if a component with size $|c| = n$ is inserted, all the original instructions following the insertion point will be shifted by n bytes, and control transfers to any of the shifted instructions need to be incremented by n.

To address the challenge of handling indirect calls and call back functions invoked by external libraries, we develop another algorithm (Section 2.5.1) that stretches a subject binary at the original entries of functions that are potential targets of indirect calls, creating small holes (usually a few bytes) to hold a long jump instruction to forward any calls to those functions to their shifted locations. These holes must not be shifted by any stretching/shrinking operations. They always stay in their original positions and we thus call them "*anchors*". Our algorithm precisely takes into account these anchors when performing stretching/shrinking. To handle indirect jumps, we leverage an efficient perfect hashing scheme to translate jump targets dynamically. We use these approaches to patch indirect jumps/calls in both the component and the target binary.

Assumptions. We make the following assumptions (and hence stating the *non-goals* of BISTRO): (1) The user, not BISTRO, will predetermine the semantic appropriateness of embedding functional component c in target program P. Furthermore, he/she will decide the specific location to insert the component. This can be practically done

by performing reverse engineering on P. For example, to harden P with some security policy enforcement mechanism based on control flow [46], the user can reconstruct the control flow graph of P, collect its dominance and post-dominance information, and decide proper locations to insert c. (2) The identification of component c in the original program Q, including its code and data, is done a priori by the user through manual or automated techniques, such as Inspector Gadget [42], binary slicing [47], binary differencing [48], and BCR [43]. While we will present our experience with functional component identification in our case studies (Section 2.7), the identification technique itself is *outside* the scope of this work. (3) Binaries can be properly disassembled (e.g., by IDA-Pro) *before* being passed to BISTRO. This assumption is supported by the large number of real-world, off-the-shelf binaries in our experiments. Although we currently do not handle obfuscated or self-modifying binaries, we note that, in addition to IDA-Pro, other conservative disassembling [10, 49] and unpacking [50] tools can also be used as the pre-processor of BISTRO to handle more sophisticated binaries.

2.3 Problems with Existing Techniques

Before presenting BISTRO, we take an in-depth look at the existing binary rewriting techniques and explain their limitations for binary component extraction and implanting. Our discussion only focuses on existing techniques that work on stripped binaries without debugging symbols or relocation information, which we classify into two categories: *detour-based rewriting* and *duplication-based rewriting*.

Detour-Based Rewriting. Detour-based binary rewriters [5–7] create control flow detours from the original code to the instrumentation. More specifically, to instrument an instruction, the rewriter replaces the instruction with a detour to a *trampoline*, where the instrumentation code is located. At the end of the instrumentation, the control flow jumps back to the original code. For example, as shown in Figure 2.2, suppose we need to count the number of times function *Func_B* gets called. The

instrumentation involves replacing the three instructions at the entry of *Func_B* with a jump instruction, which detours the control flow to the trampoline code *Tramp_B* placed at the end of the binary. *Tramp_B* will increment the counter, execute the three instructions that were replaced, and jump back to the instruction right after the instrumentation point. Detour-based rewriting works well when the number of instrumentation points is small and the instrumentation code is simple.

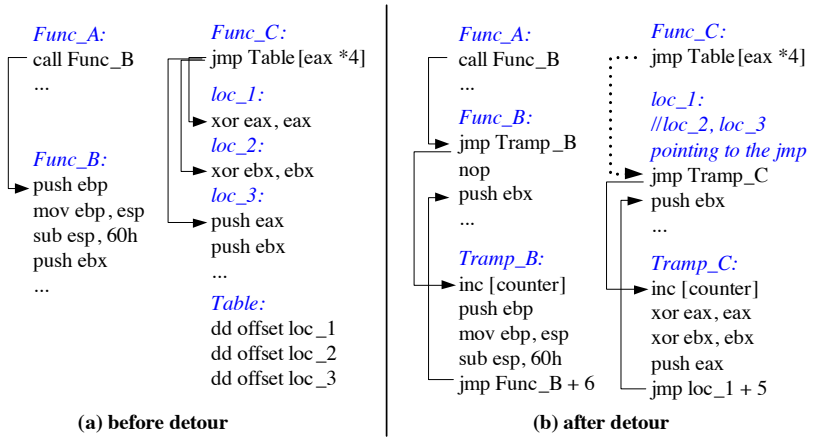


Figure 2.2.: Examples of detour using trampoline. Arrows show the direction of control-flow. The dashed line shows ill-formed control-flow.

The main problem with detour-based rewriting is that it requires the instructions to be replaced at the instrumentation point be *relocatable*. To understand this problem, let us look at the case where we try to perform the same instrumentation at *loc_1* in function *Func_C*. There is an indirect jump in *Func_C* using a jump table *Table*, with potential targets *loc_1*, *loc_2*, and *loc_3* – such structure is often generated by the compiler to represent a `switch` statement. We replace the instructions at *loc_1* with the unconditional jump instruction, which will take 5 bytes, and the instructions being replaced will be relocated to the trampoline code in *Tramp_C*. However, now *loc_2* and *loc_3* will point to the middle of the jump instruction, causing ill-formed control flow. While it seems that one can patch the jump table in this example, the problem becomes much more difficult to fix if an overwritten instruction is the computed target of some indirect jump/call, as the target may be stored in some data

structure fields or generated dynamically via complex computation. One might also consider using software breakpoint (a special one-byte instruction) instead of jump instruction to detour the control flow. However, software breakpoints incur significant performance overhead.

Duplication-Based Rewriting. Recently, duplication-based binary rewriters [8–10] are proposed. These techniques make a copy of the original code sections and then instrument the copy. The instrumented copy is executed in cooperation with the original code. In particular, to preserve control flow correctness, branch targets in the instrumented copy need to be patched. To handle indirect calls and jumps, jump/call targets *in the original code sections* are replaced with redirection to their new targets in the instrumented copy. The original data sections are also reused by the new copy.

The first problem with duplication-based techniques is their excessive space requirement. For the code sections, the space has to be almost *doubled.* Second, it is difficult to precisely determine all the possible targets of each indirect jump/call before instrumentation [9, 10] (for the sake of inserting redirection). Using a conservative analysis may result in large sets of potential targets, leading to runtime inefficiency [9].

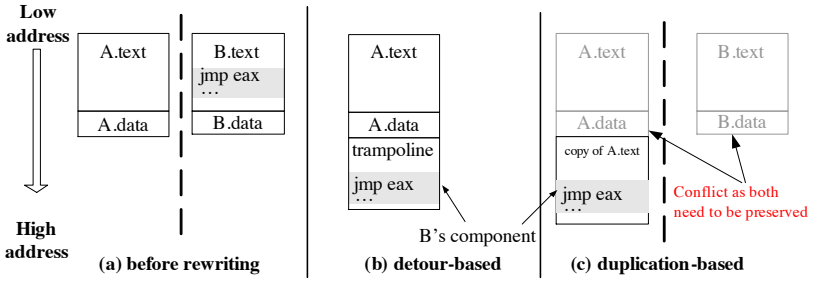


Figure 2.3.: Difficulties when transplanting a component (marked as shaded) from binary B into binary A.

Most importantly, neither duplication-based nor detour-based rewriting supports binary component transplanting – the main application scenario of Bistro. Consider the example in Figure 2.3. Suppose we wish to extract a component from B and insert

it into A. The component is shaded in (a) and contains an indirect jump instruction. In (b), a detour-based technique is applied and the component is inserted in the trampoline at the end of A's body. However, the indirect jump in the component will not work properly, jumping to some irrelevant location in A instead of to the correct target as if in B. In (c), a duplication-based techniques is applied. The text section of A is duplicated and the component is inserted to the replica. However, to ensure correctness of the indirect jump in A, it is necessary to preserve B's original text section *at the same location as in B* and insert redirection at the original possible targets of the indirect jump. Unfortunately, the position of B's text section conflicts with that of A's in the virtual address space. Such conflict is highly likely to happen in practice: by default, common compilers choose to select the same base loading address when generating executable binaries: 0x400000 for Windows PE binaries and 0x8048000 for Linux ELF binaries. This means that most binaries will overlap from the very beginning when loaded into memory.

2.4 Basic Algorithm for Binary Extraction/Stretching

In this section, we present the basic algorithm (Algorithm 1) executed by both the binary extractor and stretcher of Bistro (Section 3.3). For the time being, we assume (1) there is no indirect control transfer and (2) global data is directly referenced in an instruction using its address.

The algorithm takes the subject binary and a list of virtual address intervals called *snippets* representing (1) the holes to be created in the binary in the case of stretching *or* (2) the unwanted instruction/data blocks in the case of shrinking (extraction). First, for each byte in the binary, the algorithm computes a mapping between its original index in the binary and its corresponding index after the snippets are inserted/removed. After that, the algorithm patches address operands in control transfer and global data reference instructions, and copy each byte to its mapped location according to the mapping.

Practical Challenges. To make BISTRO work for real-world large-scale software, we still need to overcome a number of practical challenges *not* addressed by Algorithm 1.

- The target of an indirect control transfer instruction (e.g., `call eax`) is computed during execution and takes different values depending on the execution path. Such an instruction cannot be patched by Algorithm 1.

Algorithm 1 Basic binary stretching/shrinking algorithm

Input:	P – the subject binary; it has *size* and *base_addr* fields to represent its size when loaded into memory and base loading address, respectively.
	M – a list of address intervals represent code/data to be inserted/removed, sorted increasingly by their location; each interval has *addr*, *len* and *type* fields, denoting the location, size and type respectively. Type "INSERT" means inserting right before *addr*; "REMOVE" means the block starting at *addr* is to be removed.
Output:	P' – the stretched/shrunk binary.

```
1: function BasicStretching(P, M)
2:   map ← ComputeMapping(P, M)
3:   P' ← PatchTarget(P, map)
4: end function

5: function ComputeMapping(P, M)
6:   offset ← 0
7:   m ← M.begin()
8:   for i ← 0 to P.size do
9:     if m.addr == P.base_addr + i then
10:      if m.type == INSERT then
11:        offset ← offset + m.len
12:      else if m.type == REMOVE then
13:        offset ← offset − m.len
14:        i ← i + m.len
15:      end if
16:      m ← M.next()
17:    end if
18:    map[i] ← i + offset
19:  end for
20:  return map
21: end function

22: function PatchTarget(P, map)
23:   P' ← {nop, nop, ..., nop}
24:   for i ← 0 to P.size do
25:     if map[i] ≠ ⊥ then
26:       if P[i] is instruction then
27:         ins ← P[i]
28:         for each data address operand op in ins do
29:           target ← op.addr − P.base_addr
30:           off ← map[target] − target
31:           op.addr ← op.addr + off
32:         end for
33:         if ins is near call/jump then
34:           target ← i + ins.len + ins.target
35:           off ← map[target] − target
36:           off' ← map[i + ins.len] − (i + ins.len)
37:           ins.target ← ins.target + off − off'
38:         else if ins is far call/jump then
39:           target ← ins.target − P.base_addr
40:           off ← map[target] − target
41:           ins.target ← ins.target + off
42:         end if
43:         P'[map[i]] ← ins
44:       else if P[i] is data then
45:         P'[map[i]] ← P[i]
46:       end if
47:     end if
48:   end for
49:   return P'
50: end function
```

- Function pointers may be present in data or in an instruction as an immediate operand. These function pointers might be passed as parameters to external libraries as callback functions. If a function is relocated due to stretching, the external library will call back to a wrong address. All these have to be properly handled to ensure correctness of binary stretching/shrinking.

- Accesses to global data may be via data pointers (e.g., `mov ebx, ptr_data; mov eax, [ebx+4]`). The addresses of data are not known until runtime. These instructions cannot be patched using Algorithm 1 either.

We will present our solutions to these challenges in the following sections.

2.5 Handling Indirect Control Transfer

Handling indirect jumps and calls is one of the key challenges in the design of Bistro. The difficulty is that the jump/call target cannot be known statically and thus is hard to patch. To understand the challenge, consider the example in Figure 2.4. On the left, there are three objects that are connected via pointers, with two of type `B` and one of type `A`. On the right, part of function `foo()` is presented. The function takes two parameters stored in `eax` and `ebx` denoting pointer values. These two pointers may be aliased to each other. If so, `ecx` at 0x4302B2 gets the value 0x400340 defined at 0x4302A0, and then eventually the call instruction at 0x4302BD acquires the function pointer 0x444142. However, if the two pointer parameters are not aliased, the call instruction may get a completely different target, making statically patching it difficult.

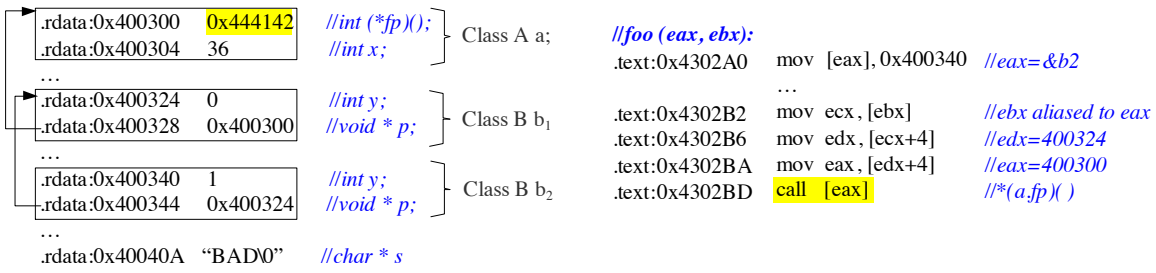


Figure 2.4.: An example showing indirect call handling in binary stretching/extraction.

A naive solution is to identify and patch any constant value in the binary that appears to be a jump/call target. But this is not safe as such values may not be jump/call targets. Notice in the example, there is a null-terminated string "BAD" at

address 0x40040A. With the little endian representation in x86, this string has the same binary value as the function pointer at 0x400300. Without type information, it is impossible to know whether the value is a string or function pointer. Failure to identify and patch a function pointer leads to broken control-flow, changing the semantics of the target binary. Misclassifying a string as a function pointer leads to undesirable changes to data. While it is plausible to leverage recent advances in binary type inference to type constants in a binary [51–54], the involvement of aliasing as in the example makes such analysis very difficult. In fact, IDA-Pro [13] failed to recognize the function pointer for this case.

If a binary has a relocation table and it does not perform any address space layout self-management such as through a packer, the relocation table will provide the positions of all constant values that are jump/call targets for BISTRO to patch them, thus lead to a sound and complete solution to binary stretching/shrinking. However, relocation table may be absent or contain bogus entries in legacy and malware binaries. Hence, in our work, *we do not assume the presence of relocation tables in our design and evaluation.* Next, we describe how to handle indirect calls in Section 2.5.1 and indirect jumps in Section 2.5.2.

2.5.1 Handling Indirect Calls

Indirect calls are very common in modern binaries to leverage the flexibility of function pointers. We have discussed the difficulty of handling function pointers at the beginning of Section 2.5. In fact, there is a more challenging situation, in which a binary may pass its function addresses to external library functions which call back the provided functions (e.g., a user function *cmp()* is provided as a parameter to an external library function *qsort()*). In this case, if a function entry has changed due to stretching or shrinking, its invocation sites are outside the body of the binary and thus beyond our control. It is difficult to patch call back function pointer parameters before they are passed on to libraries for two reasons. First, a function pointer might

not directly appear as a parameter. It could be a member of a structure passed to an external library. It may even require several layers of pointer indirection to access its value. Patching that is challenging. Second, for many external library functions, we cannot assume the availability of their prototype definitions, it is hence difficult to know their parameter types.

To handle indirect calls including call back functions, we propose to stretch the target binary to make small holes at the entry point of each function that may be an indirect call target. These holes are called *anchors*; they should not be moved during stretching/shrinking. Inside an anchor, we place a jump instruction that jumps to its mapped new address in the stretched/shrunk binary, which is the new entry of the function. As such, we do not need to identify or patch any function pointers in the binary.

Since an anchor must be placed at a fixed address in the stretched binary, it could coincide with instructions that get shifted to the address. To ensure correctness, we put a jump right before an anchor to jump over it. We call the jump the *prefix* of an anchor.

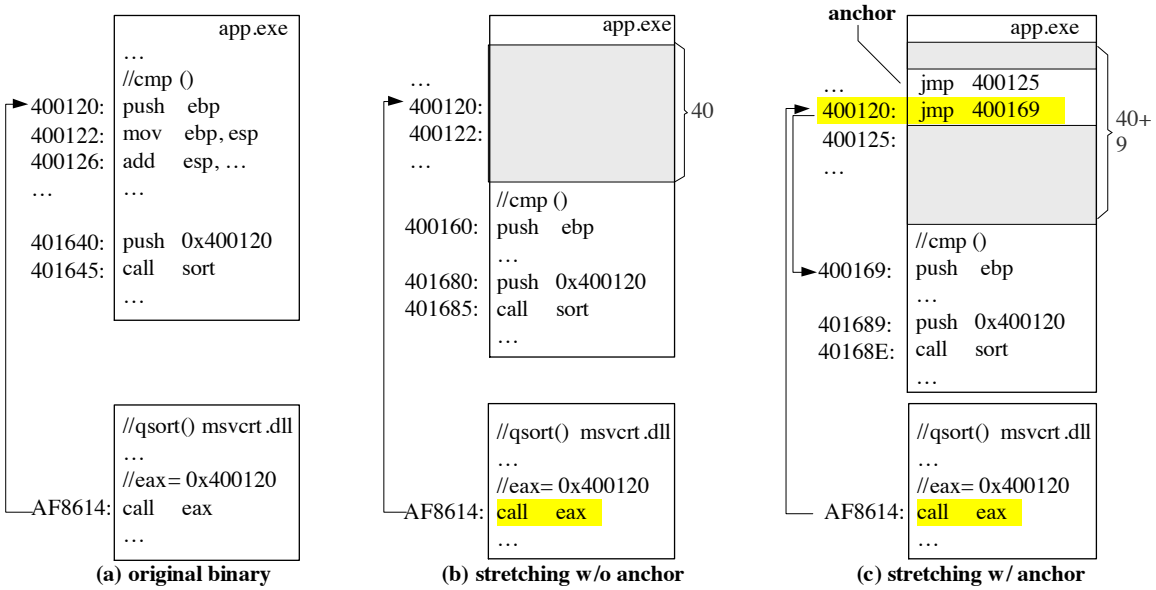


Figure 2.5.: Stretching with anchors. The shaded area in (b) is the 40-byte snippet inserted.

Consider the example in Figure 2.5 (a), in which the call-back function *cmp()* is invoked inside *qsort()*. The entry address of function *cmp()* in the original binary is 0x400120. When we stretch without anchors as shown in (b), in function *qsort()*, the indirect call to *cmp()* at 0xAF8614 will incorrectly go to 0x400120 in the shaded area. When we stretch with anchors as shown in Figure 2.5(c), an anchor containing the jump instruction will be placed at 0x400120. Any indirect call that goes to the original entry address of *cmp()*, 0x400120, will be redirected to the actual function body at the new entry address. The jump instruction preceding 0x400120 is its prefix.

Anchor-based Algorithm. With the presence of anchors, fixing control flow transfer instructions becomes more challenging than that in Algorithm 1. We hence devise a new algorithm (Algorithm 2). The idea is to divide the stretching/shrinking operation into two phases. In phase one, the subject binary program is stretched/shrunk using Algorithm 1 to create space for the inserted snippets or removed blocks. Then the stretched/shrunk binary is further stretched to insert anchors using a similar procedure. Separating the two phases substantially simplifies the interference from anchors.

Algorithm 2 Anchor-based stretching algorithm.

Input:	P – the subject binary; it has $size$ and $base\_addr$ fields to represent its size when loaded into memory and base loading address, respectively. M – a list of code/data snippets to be inserted/removed, sorted increasingly by their location; each snippet has $addr$, len and $type$ fields, denoting the location, size and type respectively. A – a list of anchors to be placed, sorted increasingly by their location; each anchor has $addr$ and len fields, denoting the location and the content size, respectively.
Output:	$anchor\_map$ – the mapping between the indices after placing snippets and their corresponding indices after anchors are placed. $prefixlen[a]$ – the prefix length of an anchor a.

```
1: function StrechingWithAnchor(P, M, A)
2:   map ← ComputeMapping(P, M)
3:   P_t ← PatchTarget(P, map)
4:   anchor_map ← ComputeAcMapping(P_t, A)
5:   P' ← PatchTarget(P_t, anchor_map)
6: end function

7: function ComputeAcMapping(P, A)
8:   offset ← 0
9:   ac ← A.begin()
10:  i ← 0
11:  while i < P.size do
12:    curaddr ← P.base_addr + i + offset
13:    if ac.addr == curaddr then
14:      prefix ← i− SIZEOF(JMP)
15:      if P[prefix] is not the start of an instruction
       then
16:        prefix ← start of instruction before prefix
17:      end if
18:      prefixlen[ac] ← i − prefix
19:      i ← prefix
20:      offset ← offset + ac.len + prefixlen[ac]
21:      ac ← A.next()
22:    else
23:      anchor_map[i] ← i + offset
24:      i ← i + 1
25:    end if
26:  end while
27:  return anchor_map
28: end function
```

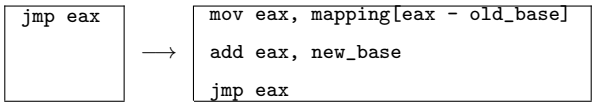
Pruning Anchors. Potentially, we can create anchors for all function entries to guarantee that we will never miss any necessary function call forwarding. However, this is not efficient. In fact, we only need to create anchors for the subset of functions that could be the possible target of some indirect call. Assuming a 32-bit machine, we construct the subset with the following criterion: *Any four-byte data value or any four-byte immediate operand in an instruction is considered a possible indirect call target, if it is equal to one of the function entries.* We obtain this subset by sequentially scanning data and code sections. Our pruning heuristic is very effective in practice. For example, the code section size of *gcc* in SPEC CPU 2000 benchmark suite is over 1MB, with over 2000 functions; after pruning, there are only 271 functions left that need anchors.

Embedding a Component with Anchors. If an extracted component contains a function that may be invoked by an indirect call in the component, Bistro will create an anchor *in the target binary* at exactly the same address of the function entry in *the component's original binary* to allow proper forwarding. If the anchor conflicts with some existing anchor in the target binary, Bistro will integrate the two overlapping anchors into an arbitration function and redirect control flow to the function instead. The function further determines which real target it should forward the call to. The calls from the target binary and those from the to-be-embedded component are distinguished by setting a flag. The arbitration function uses the flag to decide the real forwarding target.

In some rare cases, the space between two function entries might not be enough to hold the anchors. In such cases, instead of using the jump instruction for redirection, we use a software interrupt instruction, which takes only one byte. When an indirect call reaches the old function entry, a software exception will be generated and intercepted by our exception handler, which will redirect the control flow to the new function entry.

2.5.2 Handling Indirect Jumps

Indirect jumps are different from indirect calls as the jump targets may not be function entries, but rather anywhere in the binary. If we adopt the anchor approach, there would be too many anchors needed. One might leverage some heuristics such as that indirect jumps usually receive their targets from jump tables and thus simply patch the jump table entries. However, this is unsafe because of the difficulty of determining jump table boundaries. A jump table may not be distinguishable from regular data. Hence, we propose a different approach. Specifically, we insert a code snippet right before each indirect jump to translate the jump target to its mapped address in the stretched binary at runtime, as shown in the example below.



Note that the example is just for illustration. In our implementation, we use perfect hashing for address lookup, which will be explained later, and preserve the flag register during translation. Since a complete byte-to-byte mapping is computed in Algorithm 1, any indirect jump target could be properly translated and handled by this method. Observe that additional instructions need to be added to perform translation. We can easily handle this by stretching the subject binary to accommodate these instructions.

Branch Target Set Pruning. Although the translation using a complete mapping guarantees safety, it also introduces significant memory overhead. Each byte in the original binary requires 4 bytes to represent its mapped address. In fact, we only need a subset of the mapping: the stretched/shrunk binary will be safe as long as the mapping contains translation for every possible indirect jump target.

We construct the set with the following criterion: *any four-byte data value or any four-byte immediate operand in an instruction is considered a possible indirect jump target, if the value falls in the range of some code section.* We further prune the set by removing the values that point to the middle of an existing instruction. Note that

the strategy is safe for long/set jumps as their jump targets are acquired at runtime. This pruning strategy is very effective in practice. For example, the code section size of Adobe Reader X (AcroRd32.exe) is over 800KB, with over 260K instructions; after pruning, there are only 3635 possible branch targets left.

Perfect Hash Translator. The remaining challenge is to achieve fast translation. Note that after pruning, the jump target set becomes a sparse set in the address space. As a compromise between memory consumption and runtime overhead, we choose to use perfect hashing for translation. A perfect hash function maps a set of keys to another set of integer values without any collision. It guarantees $O(1)$ translation time. We use gperf [55] to generate the perfect hash function for the jump target set and compile it into a linkable .obj file that can be embedded in the target binary through Bistro.

A perfect hash function may require more space than the N keys to achieve $O(1)$ translation time. In practice, we find the size of generated perfect hash functions acceptable. For example, for the 3635 branch targets of Adobe Reader, the generated hash function is about 152KB, which is about 11% of the size of the Adobe Reader binary.

2.6 Handling Data References

Binary extraction/stretching may cause relocation of data entries, so we need to ensure the correctness of instructions referencing those data. We discuss how to address this problem from the perspectives of the target binary and the component to be embedded.

Compared to the component, the target binary is usually more complex and involves a lot of global data references. To handle this problem efficiently, we group data in the binary as continuous data blocks. If a data block might be indirectly accessed, we will make sure the block is not re-located to avoid patching data accesses, by wrapping the block in an anchor. Note that the number of data access instructions

is much larger than the number of indirect jumps/calls. Otherwise, if the data block is only directly accessed, we allow it to be relocated (by Algorithm 1). We use the following criterion: *if the value of any four-byte data, or any four-byte immediate operand (in an instruction) that is not directly used as an address falls in the range of a data block, then this block might be indirectly accessed using data pointers, and hence should not be re-located.*

In contrast, data entries extracted as part of the to-be-embedded component are most likely to be relocated. For example, if they are sparsely distributed in the address space, the BISTRO extractor (Section 3.3) will collapse them into a contiguous block, causing relocation. We adopt a method similar to the dynamic jump target translation scheme to translate data reference addresses. We add a comparison before translation to avoid translating stack or heap accesses. According to our experience, only 2% of dynamic memory references need to be translated. We further use offline static peephole scanning to identify references that surely access stack and avoid instrumenting them completely.

2.7 Evaluation

We have implemented BISTRO for Win32 PE binaries as an IDA-Pro plug-in. We have addressed a variety of engineering challenges such as virtual space layout re-arrangement with a large embedded component, patching PE header, import and export tables, and re-generating relocation table.

2.7.1 Performance: Efficiency and Overhead

We first evaluate the performance of BISTRO by stretching real-world Windows-based applications and SPEC CPU 2000 binaries. Our experiments are done on a Dell Inspiron 15R laptop with Intel(R) Core(TM) i5-2410M 2.30GHz CPU and 4GB memory, running Windows 7 SP1. For the SPEC CPU 2000 benchmark suite, we use the "win32-x86-vc7" config file which includes all integer benchmark binaries and

four floating-point benchmark binaries. We compile the benchmark suite using Visual Studio 2010, with full optimizations. To test BISTRO on non-relocatable binaries, we set "/DYNAMICBASE:NO" switch for the compiler to prevent it from generating relocatable binaries. The application binaries are readily available and we do not know about their compilers. Although the binaries of Adobe Reader and Chrome web browser carry relocation tables, we ignore them for testing our solutions for non-relocatable binaries.

We measure the following performance metrics: (1) space overhead – for both binary file and initial memory image – of a stretched binary compared with its original version, (2) runtime overhead of the stretched binary, and (3) time for BISTRO to stretch the binary. In particular, we are interested in the overhead incurred *by* BISTRO *itself*, not by the execution of the embedded components. As such, we embed a minimal component (a one-byte snippet) into each subject binary in our experiments. To create a "worst-case" scenario, we insert it at the beginning of each binary so that every byte in the binary gets shifted, which entails *all* indirect control transfer targets in the binary to be redirected. The measured overhead is hence the upper bound of overhead.

For each SPEC 2000 binary, we run both its original and stretched versions, and compare their execution time and file/initial image size. We do not measure the execution time of the Windows applications because they are all interactive. We experience no perceivable overhead when using their stretched versions.

The results are shown in Table 2.1. From the *Indirect Jumps* and *Indirect Calls* [1] columns, we observe that indirect calls are very common in application binaries, indicating that they might be C++ programs. Further investigation confirms our speculation, indicating BISTRO's effectiveness for binaries compiled from C++ programs. Moreover, there are much less indirect jumps than indirect calls, indicating they are likely to have less impact on runtime overhead. Note that a small number of indirect jumps does not imply an equally small number of potential indirect jump

[1]We exclude indirect calls to external library functions through import address table (IAT), as these external targets are not handled by our redirection mechanisms.

Table 2.1.: Performance results of stretching Windows software and SPEC CPU 2000 binaries.

Binary	Instr. Count	Indirect Jumps	Indirect Calls	Call/Jump Targets: Anchors(%)	Data Blocks: Data Anchors(%)	File Size (KB)		Initial Mem. Image Size (KB)		Run Time (s)		Stretching Time (s)
						Orig: Stch'ed	growth(%)	Orig: Stch'ed	growth(%)	Orig: Stch'ed	overhead(%)	
SPEC CPU 2000 benchmarks												
164.gzip	19825	19	103	98: 23 (23.47%)	163: 1 (0.61%)	86.5: 98.5	13.87%	424: 440	3.77%	83.2: 84.6	1.68%	0.752
175.vpr	54595	53	106	229: 31 (13.54%)	404: 1 (0.25%)	232: 248.5	7.11%	248: 268	8.06%	64.5: 64.6	0.16%	0.755
176.gcc	337033	456	260	3855: 271 (7.03%)	2580: 14 (0.54%)	1264: 1393	10.21%	1348: 1480	9.79%	33.3: 33.9	1.8%	1.420
181.mcf	20566	36	103	144: 25 (17.36%)	100: 2 (2.00%)	76.5: 85.5	11.76%	100: 108	8%	40.2: 40.4	0.5%	0.685
186.crafty	65375	56	130	312: 29 (9.29%)	247: 1 (0.40%)	283: 298.5	5.48%	1344: 1360	1.19%	38.2: 38.9	1.83%	0.935
197.parser	44554	36	112	155: 27 (17.42%)	463: 1 (0.22%)	164: 173.5	5.79%	352: 360	2.27%	83.1: 83.5	0.48%	0.754
252.eon	114249	50	441	1659: 1253 (75.53%)	1455: 1 (0.07%)	499: 575	15.23%	592: 668	12.84%	42.7: 44.7	4.68%	0.950
253.perlbmk	164093	148	211	2166: 499 (23.04%)	1293: 6 (0.46%)	626: 743	18.69%	648: 764	17.9%	63.3: 67.9	7.27%	1.118
254.gap	129464	35	1357	816: 625 (76.59%)	1142: 1 (0.09%)	452.5: 492	8.73%	896: 936	4.46%	35.4: 37.2	5.08%	1.001
255.vortex	132034	66	145	446: 71 (15.92%)	738: 1 (0.14%)	561: 585	4.28%	588: 612	4.08%	50.6: 51.1	0.99%	1.050
256.bzip2	21360	36	101	145: 25 (17.24%)	150: 1 (0.67%)	87.5: 99	13.14%	172: 184	6.98%	73.4: 74.6	1.63%	0.714
300.twolf	64669	41	106	193: 30 (15.54%)	391: 2 (0.51%)	253: 263	3.95%	296: 304	2.7%	93.2: 93.6	0.43%	0.809
177.mesa	143679	211	552	2675: 473 (17.68%)	942: 5 (0.53%)	549.5: 652.5	18.74%	568: 672	18.31%	64.9: 65.6	1.08%	0.990
179.art	23353	38	103	149: 26 (17.45%)	103: 2 (1.94%)	85.5: 94.5	10.53%	104: 112	7.69%	32: 32.3	0.94%	0.690
183.equake	21824	38	101	146: 27 (18.49%)	116: 1 (0.86%)	88.5: 97	9.6%	104: 112	7.69%	26.1: 26.1	0%	0.720
188.ammp	61214	39	128	224: 70 (31.25%)	279: 1 (0.36%)	235.5: 245.5	4.25%	252: 264	4.76%	88.7: 88.3	1.92%	0.780
Average	-	-	-	- (24.80%)	- (0.60%)	-	10.09%	-	7.53%	-	1.90%	-
Real-world Windows-based Software												
putty	107220	57	662	942: 291 (30.89%)	93: 1 (1.08%)	444: 496	11.71%	472: 524	11.02%	-	-	0.865
gvim	561626	294	5111	3893: 1004 (25.79%)	5081: 22 (0.43%)	1950.5: 2150	10.23%	2008: 2212	10.16%	-	-	2.121
notepad++	272434	159	4302	4897: 2695 (55.03%)	3394: 7 (0.21%)	1584: 1864	17.68%	1660: 1940	16.87%	-	-	1.480
Adobe Reader	273710	146	2543	3635: 2160 (59.42%)	3037: 11 (0.36%)	1445.9: 1702.4	17.74%	1472: 1728	17.39%	-	-	1.556
Chrome	230234	82	1280	1842: 933 (50.65%)	930: 6 (0.65%)	1211: 1338	10.49%	1240: 1368	10.32%	-	-	1.391
Average	-	-	-	- (44.36%)	- (0.55%)	-	13.57%	-	13.15%	-	-	-

targets. In fact, due to the difficulty of identifying jump table boundaries, we conservatively consider any constant in a binary that appears to be an instruction address as a potential jump target. The large number of potential jump targets and the low impact on performance justify our design choice of using the slightly more expensive but more flexible dynamic target translation scheme (Section 2.5.2), compared to the anchor scheme (Section 2.5.1).

The *Call/Jump Targets: Anchors* column shows the number of potential indirect call/jump targets, the number of anchors generated, and their comparison. Observe that the number of anchors created is small, compared to the size of the potential set. For binaries from C++ programs, due to the heavy use of virtual methods, it is not a surprise to see many anchors created. The *Data Blocks: Data Anchors* column shows that only less than 1% of all data blocks need to be preserved at their original locations using anchors. From the *File Size* columns, we can see BISTRO only increases the file size by 10.1% on average for SPEC programs, and 13.6% for application binaries. The overhead is dominated by the perfect hash tables. The *Initial Mem. Image Size* columns show the initial memory consumption when the binary is loaded into memory, which increases by only 7.5% on average for SPEC programs and 13.2% for application programs. Note that BISTRO does not cause any additional memory overhead during execution. The *Run Time* columns present the runtime overhead, which is only 1.9% on average. Except *eon*, *perlbmk* and *gap*, all SPEC binaries have less than 2% overhead. The last column *Stretching Time* shows the stretching time of BISTRO. The time is consistently short, implying that BISTRO can stretch a binary at runtime when it is loaded.

2.7.2 Case Study I: Binary-level Semantic Patching Using BISTRO

Code reuse is a common practice in software development. One popular approach is to directly compile and statically link a piece of re-usable code with the target software – either directly in the executable or in some private library – to make

the software self-contained, avoid compatibility problems, and improve performance. Indeed, developers of many popular programs (e.g., *chrome* and *firefox*) reuse code this way. The consequence is that programs reusing the same code may have the code placed at different locations in their address spaces. The reused code may not even have the same instructions if compiled by different compilers.

However, code reuse via static linking introduces a security liability: When a piece of re-usable code contains a vulnerability, all programs that reuse the code will suffer from the same vulnerability. If these programs have been shipped in binary forms, the only way to fix the vulnerability is to release multiple binary patches – one for each program and by the corresponding vendor. However, not all vendors react to a vulnerability with equal timeliness and some may not even be aware of the vulnerability not in their own code. Thus it may be desirable for customers, who do *not* have source code access, to patch these programs without vendors' involvement. Binary *syntactic patching*, which directly applies a patch for software A to software B sharing the same (vulnerable) code, will hardly work, because of the different locations of the code and the syntactic differences between the two code copies (due to different compilers used or different call/jump targets inside the copies).

In our first case study, we show that BISTRO can enable *binary semantic patching*. Assume that software A and B share a function f and the vendor of A has released a binary patch of f for a vulnerability. Let the patched program and the patched function be A' and f', respectively. We will use BISTRO to extract f' from A' and embed it to B to replace the vulnerable version. Note that BISTRO is critical in ensuring the extracted f' is properly patched and the target binary B is properly stretched to contain f'.

We acquire a group of application binaries that leverage the same vulnerable component using public, vendor-provided information (e.g., which libraries are used in the software) or by finding similar binary snippets using the binary comparison tool *bindiff* [48]. Suppose at least one binary in the group, say A, has a patched

Table 2.2.: Results of binary semantic patching using Bistro.

Vulnerability	Patch Extracted From	Vulnerable Application Patched	Original File Size (KB)	Patched File Size (KB) w. / w.o. Reloc	Semantic Patch Available	Vendor Patch Available
CVE-2010-1205	libpng 1.2.43 → 1.2.44 (rpng2-win.exe)	Firefox 3.6.6 (xul.dll)	11747.5	12371.5 / 13005	6/25/2010	7/20/2010
CVE-2011-3026	libpng 1.4.8 → 1.4.9 (rpng2-win.exe)	Zoner Photo Studio 15 (Zxl.dll)	8225.1	8502.1 / 9181.6	2/18/2012	N/A
SA47322 / CVE-2012-0025	IrfanView 4.30 → 4.32 (Fpx.dll)	XnView 1.99.5 (Xfpx.dll)	356	368 / 400	12/20/2011	N/A
		LeadTools 17.5 (ltkdku.dll)	138.5	143 / 151	12/20/2011	N/A
SA47388	XnView 1.98.5 → 1.98.8 (Xfpx.dll)	IrfanView 4.35 (Fpx.dll)	432	448 / 508	3/12/2012	N/A
		LeadTools 17.5 (ltkdku.dll)	372.5	428.5 / 493.5	3/12/2012	N/A
SA48772 / CVE-2012-0278	IrfanView 4.33 → 4.34 (Fpx.dll)	XnView 1.99.5 (Xfpx.dll)	356	368 / 400	4/13/2012	N/A
		LeadTools 17.5 (ltkdku.dll)	138.5	142.5 / 150.5	4/13/2012	N/A
SA49091	XnView 1.98.8 → 1.99 (Xfpx.dll)	LeadTools 17.5 (ltkdku.dll)	372.5	428.5 / 488.5	6/15/2012	N/A

version A'. Our goal is to extract a *semantic* patch out of A' and transplant it to patch the other vulnerable binaries $\{B_1, ..., B_n\}$.

We collect 6 real-world vulnerabilities, with their CVE or Secunia IDs shown in Column 1 of Table 2.2. For each vulnerability, the vulnerable program(s) that has been patched by its vendor is shown in Column 2. The file names in braces represent the files that are patched. Column 3 shows a list of other un-patched programs with the same vulnerabilities. Column 6 shows the patch release date for the application in Column 2, i.e. the earliest date we can extract the semantic patch. Column 7 shows the date when the vendors for the software in Column 3 release their patches (N/A means no vendor patch is available yet). Most of the applications used in this case study are close-source (except *libpng* and *firefox*). Observe that most of the applications in Column 3 do not have vendor patches so far. For *firefox*, the new version (3.6.7) which patched the vulnerability was released – but with a one-month latency. With BISTRO, we can fix all these vulnerable applications as soon as one vendor releases the corresponding patch.

Failure of Syntactic Patching. We first verify that simple syntactic patching does not work – that is, using an existing binary differencing tool that generates and applies patches (e.g., *xdelta*, *bsdiff*, *bspatch*, etc.) will not properly patch $B_{1...n}$. For each vulnerability in Table 2.2, we use *bsdiff* to extract the syntactic difference between the pair of shared functions (f and f') in the versions in Column 2 as a patch, and use *bspatch* to apply it to the corresponding vulnerable applications in Column 3. None of the resultant binaries works. Further inspection shows that syntactic patches cannot properly fix the call/jump targets that are different among copies of the same reused code.

Function Identification. To extract the semantic patch for a specific vulnerability, we need to identify the functions in A and A' that are related to the vulnerability. To illustrate, we denote the set of functions in a binary A by F_A. First, we notice that the related functions must exist in all the vulnerable binaries. We take A and the other vulnerable binaries $\{B_1, ..., B_n\}$ and use *bindiff* [48] to identify the set of common

functions F among them (Equation (2.1) below). However, some functions in F are not related to the vulnerability (e.g., other pieces of reused code.) To pinpoint the relevant functions in F, we leverage the observation that they have been patched in A'. Particularly, we utilize the partial matching feature of *bindiff* to identify the relevant patched functions as shown in Equation (2.2). The generated M is a mapping that maps a function $f \in F$ to its patched version $f' \in F_{A'}$. Two functions are said to be partially matched when they share similar characteristics (e.g., common basic blocks, similar CFG) but are not exactly the same. By performing partial matching between F and $F_{A'}$, we also exclude patches in A' that are not related to the target vulnerability. Note that a vendor may patch a few (unrelated) problems in a single release.

$$F = F_A \cap F_{B_1} \cap F_{B_2} ... \cap F_{B_n} \tag{2.1}$$

$$M = BinDiff\_Partial\_Matching(F, F_{A'}) \tag{2.2}$$

Patch Transplanting. We have developed a binary semantic patching tool based on BISTRO and *bindiff*. The extraction and application of the patch is guided by mapping function M. For each mapping under M: $(f, f' \in F_{A'})$, we use BISTRO to extract f' from A' as the semantic patch for f. For each vulnerable binary B, we use *bindiff* to find f. We use BISTRO to cut out f and then stretch the resulting binary to implant f' at the same starting address of f. BISTRO ensures the correctness of both f' and the patched binary B' by properly stretching and patching control transfer instructions and data references. Our patching tool tries to avoid extracting dependent functions or global data entries of f' (i.e., functions being called and global data accessed by f') as much as possible by redirecting them to their counterparts in the target binary B. Since f' is a patched version of f, they likely share the same dependencies. For example, for each function invocation to function g' inside f', if *bindiff* is able to identify the matching function g in B, our tool will automatically redirect the invocation in the extracted patch to g, without extracting g'. To be

conservative, g and g' must be fully matched. Otherwise, g' will be extracted as part of the semantic patch.

We evaluate our patching tool on the subjects in Table 2.2. We apply our tool in two different ways to stress-test the robustness of BISTRO: first, we use the relocation information when it is present in the binary; second, we do not use relocation information at all. The patches are not large, each consisting of tens to hundreds of instructions. However, it is not straightforward to generate them because of the nature of the vulnerabilities being patched. In both runs, the patching is successful: the patched applications work well and no longer suffer from the corresponding vulnerabilities. Columns 4 and 5 show the file size changes.

The first two vulnerabilities are in *libpng*, which is widely used in various software to read, write and render PNG images. The two vulnerable applications in Column 3 have *libpng* statically linked in their private DLLs (*xul.dll* and *Zxl.dll*). To patch these DLLs, we extract the semantic patch from *rpng2-win.exe*, a sample application in the *libpng* package. The remaining four vulnerabilities lie in *libfpx*, a library to handle the Flashpix (.fpx) image format. For the four vulnerabilities, only the first one was patched by the maintainer of *libfpx*; the other three were patched by individual developers who use *libfpx*. However, as shown in the table, individual developers only care about patching the *libfpx* code in their *own* applications. Using our binary semantic patching tool, users of the un-patched applications can transplant the patches and eliminate the vulnerabilities without the help of application developers.

2.7.3 Case Study II: Malware Stitching Using BISTRO

In the second case study, we demonstrate how BISTRO helps in the study of cyber attacks and counter-attacks. Specifically, we use BISTRO to compose a new, *executable* malware by stitching 3 separate functional components extracted from a *non-executable* sample of the Conficker worm [45]. It is an *unpacked* version without relocation information. Based on the published technical report of Conficker [45] and

manual code inspection, we identify the code and data associated with the following 3 components:

- **DNS API hijacking.** This component prevent DNS query of the web sites in a blacklist by hijacking the functions *Query_Main*, *DNSQuery_A*, *DNSQuery_W* and *DNSQuery_UTF8* in *dnsapi.dll*. The result is these web sites will not be accessible using their domain names.

- **Code injection.** To hijack the functions in *dnsapi.dll* used by a process (e.g., Internet Explorer), the malware must inject itself into the address space of the process. This component performs the injection. It takes the process identifier (PID) of the target process and the path of the malware as parameters.

- **Process identification.** This component gets a process' PID using its process name and provide the PID to the code injection component.

The identification process takes us about 60 minutes. After that we use BISTRO to extract the three components from the Conficker sample. We then create a dummy DLL to serve as the container of these components. Next, we use BISTRO to embed the 3 components into the empty DLL, right before the *DllMain()* function. After that, we add instructions to the *DllMain()* function to invoke the inserted components. The invocation code first checks if the current process is the target process. If so, it will invoke the DNS API hijacking component to hijack the DNS query. If not, it will call the process identification component to find the PID of the target process, and then call the DLL injection component to inject itself into the target process for DNS API hijacking. The whole composition process takes us about 30 minutes.

To verify the functionality of the newly composed malware, we select two applications as our targets (in two experiment runs): Internet Explorer and FlashFXP (an FTP client). After being loaded, the malware injects itself into the target processes. Then, in the target application, we try to access web site *avast.com*, which is blacklisted by Conficker [45]. Interestingly, the access was not blocked at first (namely,

the malware did not succeed). After debugging, we found that it was due to a bug in Conficker's original code: the hijacked *DNSQuery_W()* has one unnecessary instruction which sets a wrong return value. We point out that *we would not have spotted the problem, had we not made these components executable and observed their runtime behavior.* After removing this instruction using BISTRO, both IE and FlashFXP are successfully compromised: they can no longer access *avast.com* due to a DNS query error.

2.7.4 Case Study III: Trojan-ing Kernel Drivers

Table 2.3.: Trojan-ed device drivers (two per row).

Original Driver	File Name	File Size(KB)	w/ proc_hider embedded		w/ keylogger embedded	
			File size(KB)	Work?	File size(KB)	Work?
Beep	beep.sys	4.1	7.9	√	12.4	√
FAT File System	ftdisk.sys	122.1	135.1	√	137.5	√
NT File System	ntfs.sys	561.1	595.3	√	598	√
Intel E1000 Network Adapter	e1000325.sys	167.1	175.4	√	180.6	√
Logitech C500 Webcam	LVPr2Mon.sys	25	31.4	√	33.8	√

In the third case study, we demonstrate the use of BISTRO in transplanting malicious modules from existing kernel rootkits to existing kernel-level device drivers. The trojan-ed kernel drivers will execute the rootkit modules while performing their original functionalities. The goal of this case study is two-fold: (1) to evaluate the effectiveness of BISTRO for kernel-level binaries, (2) to show the possibility and ease of composing – instead of implementing from scratch – device drivers with hidden and possibly malicious logic. Such trojan-ed device drivers are more difficult to detect and clean up, compared with traditional rootkits that come as stand-alone kernel modules. On the flip side, trojan-ed kernel drivers can also be leveraged in defensive missions, such as honeypot deployment, to achieve better stealthiness in attack monitoring and containment. For example, malware may try to aggressively detect and disable any monitoring kernel module (e.g., Sebek). With BISTRO, one could transplant stealthy

monitoring/logging functions into a general-purpose device driver, making them more difficult to detect and disable.

In this case study, the two Windows-based kernel rootkits tested are captured variants of *HookSSDTMDL* and *Klog* which were originally packed in the wild, without relocation information for the rootkit code. The packers use their own algorithms to perform rootkit code relocation, and such relocation information is lost after the rootkits are unpacked. The samples we obtained are the *unpacked* version. We wish to extract two modules from the samples (one from each): (1) *proc_hider* for hiding processes and (2) *keylogger* for logging keystrokes. To show the generality of device driver trojan-ing, we transplant these two rootkit modules into 5 different Windows-based kernel drivers, resulting in a total of 10 trojan-ed kernel drivers.

First, we use an approach similar to [44, 47] to identify the modules to extract. We then use Bistro to shrink each of the two kernel rootkits so that only the code of the two modules and the data they access remain (as snippets). The size of the extracted snippets is 2.3KB for *proc_hider* and 7KB for *keylogger*, respectively. The size of the data in the extracted snippets is 169 bytes and 514 bytes, respectively. After preparing the snippets, we use Bistro to insert each of them into each of the following five drivers: *beep.sys, ftdisk.sys, ntfs.sys, e1000325.sys* and *LVPr2Mon.sys*. The OS is Windows XP (SP2)[2]. Table 2.3 lists the 10 resultant trojan-ed drivers (two per row). For each of the 10 drivers, we install it and confirm the proper working of (1) the original driver functionalities and (2) the malicious rootkit module.

When determining where to insert a rootkit module, we choose to insert it right before a randomly chosen function in the driver. To invoke the rootkit module when the driver is loaded, we insert a call to the rootkit module in the `DriverEntry ()` function of the driver, which is a mandatory function exported by any driver and can be located in the code section by reading the export table. Interestingly, we are able to use Bistro to implement a "timebomb"-style invocation of the rootkit module: Instead of activating the module upon driver loading, we wish to invoke it only under

[2]We use Windows XP because the real-world rootkits we obtained do not work with newer versions.

a certain condition. Specifically, when we trojan the NT file system driver (`ntfs.sys`) with the keylogger module, we want to activate the keylogger only when a file with the word "secret" in its name is opened. This is done by calling a file-name-matching function before activating the keylogger. We write this function using C, compile it into a binary snippet, and use BISTRO to insert it into `ntfs.sys` (just like inserting the rootkit module), particularly, inside `NtfsFsdCreate ()`, a function that is called every time a file is opened. Here, we leverage IDA-Pro to spot this function, which is the IRP dispatch routine to handle `IRP_MJ_CREATE` IRPs. This can be easily done by finding the initialization of the IRP dispatch table in `DriverEntry ()`. We verify that the timebomb-controlled trojan-ed driver works as expected.

We observe that, for a "native" driver developed by the OS vendor (e.g., *beep.sys, ftdisk.sys* and *ntfs.sys*), the installation of the trojan-ed version does raise an alert to the user, thanks to the built-in integrity check mechanism in the OS. Unfortunately, if the user chooses to ignore the alert, the installation will proceed and the system will never complain again. For third-party drivers (e.g., *e1000325.sys* and *LVPr2Mon.sys*), the detection of maliciously trojan-ed version is much more difficult because these drivers may be widely distributed and frequently updated without a centralized authority. Even if such an authority exists and performs digital signing for its drivers, authors of trojan-ed drivers may still evade detection by stealing certificates from the authority to sign their trojan-ed drivers, as was done in the crafting of Stuxnet [56]. In our study, the installations of the trojan-ed third-party drivers did not trigger any warnings.

2.8 Limitation

BISTRO cannot work on self-modifying, self-checking or obfuscated binaries. Self-modifying binaries generate instructions dynamically during runtime, which could not be statically patched using BISTRO. Self-checking binaries use checksum or other integrity checks to detect changes made to their code by BISTRO, thus may refuse to

run properly. Obfuscated binaries in many cases cannot be properly disassembled. For instance, the attacker can craft a conditional jump, with one branch never taken but pointing to a data entry. A disassembler will have trouble handling such binaries as it does not know statically that one of the branches cannot be taken. However, we note that all other static binary rewriting/instrumentation techniques face the same challenge.

Our anchor and branch target set pruning criteria assume the constants in a binary represent a superset of all possible indirect control transfer targets. This assumption should hold for binaries generated by common compilers. One exception is position independent code (PIC), which obtains addresses at runtime and use them to compute indirect control transfer targets. All PIC we encountered has the form of making a call and then obtain the return address from the stack (e.g., *call $+5; pop eax*), which is the address of the instruction right after the call. We identify all such instructions and insert snippets to adjust the addresses to their mapped addresses. Also, special compilers or hand-written binaries might violate our assumption. For example, in the instruction sequence *mov eax, Target; add eax, 5; jmp eax*, the actual target is *Target* +5 instead of the constant *Target*; our pruning heuristic will miss the actual target. For such binaries, we can choose not to prune the anchor set or the branch target set, which will consume more memory but guarantee correctness.

2.9 Related Work

The most related work is discussed in Section 2.1 (with details in Section 2.3.) In this section, we discuss other related work in the general area of binary manipulation.

Static Binary Rewriting. Static binary rewriting is widely applied in many scenarios, such as in-lined reference monitors [57], software fault isolation [46,58–60], binary instrumentation [1, 2, 4, 6–8], binary obfuscation [61, 62] and retrofitting security in legacy binaries [9, 63]. Most of these rewriters require the binary to be compiled by specific compilers, or contains symbolic information.

PEBIL [8], REINS [57], STIR [10] and SecondWrite [9] are recently developed rewriters targeting stripped binaries. However, they all aim at rewriting a single binary, so they all keep the original code and data sections in place. In contrast, BISTRO supports "transplanting" binary components from one or more binaries to a target binary, which requires rewriting and combining multiple binaries. Keeping original code and data sections in place may result in address space conflicts and hence is not an option for BISTRO. Detour-based techniques [5–7] are lightweight and can work on stripped binaries. However, they cannot patch non-trivial jumps/calls that are repositioned.

Dynamic Binary Rewriting. Dynamic binary rewriters [17,19,20,41] are generally more robust as they do not require specific compilers or symbolic information. It is possible to apply them to conduct binary stretching and transplanting. However, we choose to use a static approach mainly because of the following two reasons: (1) Dynamic binary rewriters usually have much higher run time overhead than static ones. (2) It is more difficult to deploy a instrumented binary using dynamic approaches, as the rewriter itself must be deployed along with the binary.

Binary Component Identification, Extraction and Reuse. Researchers proposed to identify, extract and reuse components from binaries for security scenarios [42–44]. Kolbitsch et al. proposed Inspector Gadget [42], which performs dynamic slicing on a malware binary to identify and extract the slice pertinent to a specific malicious functionality, and wrap the slice into a stand-alone binary that could be reused to execute the malicious functionality. Inspector Gadget is able to extract component from self-modifying code, which is not supported by BISTRO due to the limitation of static binary manipulation. Using dynamic slicing, Inspector Gadget also avoids the problem of handling indirect calls/jumps in BISTRO as all call/jump targets are directly known in the slice. However, the slice may not cover all possible code paths, which could result in incorrect execution when the user provides an input that would lead to a code path which is not included in the slice. Compared to

Inspector Gadget, BISTRO statically extracts the component from the binary, which involves handling of indirect calls/jumps but provides better code path coverage.

Caballero et al. proposed BCR [43] to identify and extract a function from a binary using a combination of static and dynamic analysis. The extracted function, in the format of disassembly, is wrapped in a C file to be reused. BCR statically disassembles the designated function starting at its entry point; when encountering indirect call/jumps, BCR utilizes dynamic execution trace to the find the call/jump targets. During the extraction, BCR rewrites all calls/jumps to use labels. Using labels implies that indirect call/jump can only have one target, which may not always hold in practice. Although BCR specially handles indirect jumps that use jump tables, there are other forms of multiple-target indirect calls/jumps such as function pointers and vtables. Compared to BCR, BISTRO preserves the original semantic of indirect calls/jumps when performing the extraction, hence does not suffer from this problem.

Neither Inspector Gadget nor BCR could extract components from non-executable binaries (as in Section 2.7.3) because they are based on dynamic analysis. In such case, BISTRO can still perform the extraction statically. Moreover, neither Inspector Gadget nor BCR supports reusing extracted components to enhance legacy binaries (as in Section 2.7.2), as they lack the capability of embedding instructions that invoke the components into the target binary. BISTRO is able to handle such a scenario by performing both binary component extraction and embedding.

Lin et al. proposed ROC [44] which uses dynamic slicing to identify reusable functional components in a binary for attack purposes. However, compared to BISTRO, ROC only reuses the components in the same binary; it does not support extraction or reusing components in a different program.

2.10 Summary

In this chapter, we have presented a new pair of binary program manipulation primitives called BISTRO for extracting and re-packaging a functional component

from a binary program; and for embedding a functional component in a target binary program, respectively. We address the challenges of patching control transfer instructions and data references to preserve the semantics of both the extracted component and the stretched binary program, especially indirect calls and jumps. BISTRO incurs low runtime overhead (1.9% on average) and small space overhead (11% on average). The extraction and embedding operations are highly efficient, with less than 1.5s for most cases. We have applied BISTRO to three security application scenarios, demonstrating its efficiency, precision, and versatility.

3 SPIDER: STEALTHY BINARY INSTRUMENTATION VIA HARDWARE VIRTUALIZATION

3.1 Introduction

In a wide range of security scenarios, researchers need to trap the execution of a binary program, legitimate or malicious, at desired instructions to perform certain actions. For example, in high accuracy attack provenance, instruction-level trapping allows recording of events which are more fine-grained than system calls and library calls. In malware analysis, where malware often includes large number of garbage instructions to hamper analysis, it allows analysts to skip such instructions and focus on the instructions that are related to the behavior of malware.

Debuggers [12–14] and dynamic instrumentation tools [17–22] both support efficient instruction-level trapping. As a countermeasure, an increasing percent of malware is equipped with anti-debugging and anti-instrumentation techniques. Such techniques are also commonly used in legitimate software for protection purpose [64]. While they do prevent reverse-engineering and software modification, they also render any security application that relies on instruction-level trapping infeasible at the same time.

Researchers have proposed to build systems that enable transparent trapping to solve the problem. However, existing approaches are insufficient to support transparent, efficient and flexible instruction-level trapping. In-guest approaches [15, 16] could be detected by program running in the same privilege level. Emulation based approaches [25, 26] are not transparent enough due to imperfect emulation. Hardware virtualization based systems [27, 50, 65–67] provide better transparency. However, none of them supports instruction-level trapping with both flexibility and efficiency. Some of them utilize single-stepping which results in prohibitive performance over-

head; others could trap only a certain subset of instructions. More detailed discussion about existing work is presented in Section 4.7.

In this chapter, we present SPIDER, a stealthy program instrumentation and debugging framework built upon hardware virtualization. We propose a novel primitive called *invisible breakpoint* to support transparent, efficient and flexible trapping of execution at any desired instruction in a program. Invisible breakpoint is an improvement over traditional software breakpoint, with all its side-effects hidden from the guest. SPIDER hides the existence of invisible breakpoint in the guest memory by utilizing the Extended Page Table (EPT) to split the code and data view seen by the guest, and handles invisible breakpoint at the hypervisor level to avoid any unexpected in-guest execution. SPIDER also provides data watchpoint which enables monitoring memory read/write at any address.

We have developed a prototype of SPIDER on KVM [68]. We have evaluated the transparency of SPIDER using software protectors and programs equipped with state-of-the-art anti-debugging and anti-instrumentation techniques. The result shows that SPIDER successfully maintains transparency against all of them. We have also applied SPIDER to the following cases: (1) We improve the applicability and security of an existing attack provenance system [11] by replacing its underlying in-guest instrumentation engine with SPIDER; (2) We demonstrate a threat that involves stealthy introspection on protected software to capture sensitive application data. The performance overhead introduced by SPIDER is less than 6% in our case studies. The quantitative cost of each trap is around 3200 CPU cycles according to our measurement, which is less than a previous work [15] and comparable with hardware breakpoint.

3.2 Related Work

In this section, we take an in-depth look at existing program debugging, instrumentation and analysis tools and discuss their limitations. We only focus on

instruction-level tools as they are most related to SPIDER. We classify them into four categories: *in-guest*, *emulation based*, *hardware virtualization based* and *hybrid*.

In-Guest Approaches. Traditional in-guest debuggers [12–14] use software and hardware breakpoints to gain control at arbitrary points during the execution of a program. In x86, software breakpoint is implemented by replacing the target instruction with a special 1-byte instruction (*int3*), which triggers a #BP exception upon its execution. Hardware breakpoints are implemented as four debug registers (*DR0-DR3*). Each of these registers holds a target address; a #DB exception is triggered upon instruction execution or data access at the target address. Software breakpoints could be easily detected by code integrity checks as the instruction is modified. Hardware breakpoints are not transparent either. The reason is that they are limited resource such that programs could hold and use all hardware breakpoints exclusively to prevent debuggers from using them.

To solve the transparency issue of traditional breakpoints, researchers proposed to use page-level mechanism to trap execution of arbitrary instruction [15, 16]. The page which contains the target instruction is set to non-present, which will cause a page fault upon execution. In the page fault handler, the page is set to present and the target instruction is executed in single-step mode. Then the page is set back to non-present to enable breakpoint again. There are two limitations with this approach. First, execution of any instruction in the non-present page will cause a page fault, even if there is no breakpoint set on that instruction. This would result in prohibitively high performance overhead. Second, although it is not as straightforward as detecting traditional breakpoints, the modified page table and page fault handler could still be detected by kernel-level programs.

Dynamic binary instrumentation (DBI) frameworks [17–22] are able to insert instrumentation code at arbitrary points during the execution of a program. The mechanism of DBI frameworks is to relocate and instrument code blocks dynamically and handle control flow transitions between basic blocks. Transparency is an important concern in DBI frameworks. For example, position-independent code makes assump-

tion about relative offsets between instructions and/or data. DBI frameworks may break such assumptions when relocating basic blocks, so they must change some instructions in the program to create an illusion that every address is the same as in a native run. However, despite recent efforts [23, 24] targeting at improving the transparency of DBI frameworks, they are still insufficient. A recent work [69] has also shown that there are a number of ways to detect DBI frameworks. More essentially, the DBI framework itself, along with the relocated and instrumented basic blocks must occupy additional memory in the virtual address space. Programs could scan the virtual address space to detect unsolicited memory consumption and hence the DBI framework.

Emulation Based Approaches. To get rid of in-guest components that are visible to guest programs, researchers have proposed to build program analysis and instrumentation tools [25, 26] using full system emulators such as QEMU [41] and Bochs [70]. Full system emulators create a virtual environment for the guest so it feels like running in a dedicated machine. Instruction-level trapping could be easily implemented as each instruction is emulated. However, attackers have been able to identify various methods [71–73] to detect emulators by exploiting imperfect emulation of instructions and hardware events (e.g. interrupts and exceptions). Although imperfection that is already known could be fixed, the problem still exists as long as there might be unrevealed imperfections. In fact, it has been proved in [50] that determining whether an emulator achieves perfect emulation is undecidable.

Hardware Virtualization Based Approaches. With recent advances in processor features, researchers propose to leverage hardware virtualization to construct more transparent program analysis and instrumentation tools [27, 50, 65–67]. Hardware virtualization naturally provides better transparency than emulation by executing all guest instructions natively on processor.

Among existing hardware virtualization based approaches, none of them supports transparent, efficient and flexible trapping of arbitrary instructions during execution of a program. PinOS [27] implements a DBI framework on the Xen [74] hypervisor.

As it needs to occupy part of the guest virtual address space, it suffers from the same transparency issue as in-guest DBI frameworks. Ether [50] and MAVMM [65] use single-stepping for instruction-level trapping, which triggers a transition between hypervisor and guest upon execution of every guest instruction. Such transition causes significant performance overhead as it costs hundreds to thousands cycles while an instruction only costs several to tens cycles on average. The mechanism is not flexible either as one is forced to single-step through the whole program even if he is only interested in the states at specific points during execution. Such scenario is often encountered when analyzing obfuscated programs, which contain lots of garbage code.

Several recent approaches [66,67,75] propose to use x86 processor features to trap specific events for program analysis. In [66], the authors use branch tracing to record all the branches taken by the program during its execution. While the performance is much better than single-stepping, it is still 12 times slower than normal execution. Also, the tool is only able to record all branches. It cannot trap a specific branch, which renders detailed analysis at arbitrary given points during execution impossible. In [67], the authors make use of performance monitoring counters (PMCs) to trap certain types of instructions (e.g. call, ret and conditional branches). However, there are still many other types of instructions (e.g. mov) that could not be trapped this way. Also, the tool does not support trapping instruction at a specific location. In [75], the authors propose to utilize the System Management Mode (SMM) in x86 to implement a debugging system with maximized transparency. However, when performing instruction-level debugging, their system introduces more than 900 times slowdown compared to a native execution, which is only usable for manual debugging instead of instrumentation in production runs.

Hybrid Approaches. Researchers have also proposed to use hybrid approaches [28, 29] to take advantage of both the transparency granted by hardware virtualization and the flexibility provided by emulation. In [28], the authors utilize the trace obtained from a transparent reference system (e.g. Ether) to guide the execution of program in an emulator. However, as discussed above, it incurs high performance overhead

to obtain execution trace using current hardware virtualization based approaches. V2E [29] takes another approach by emulating only the instructions that can be perfectly emulated. For other instructions in the program, it records the state changes caused by these instructions in a hardware virtualization based system, and then replays the state changes in the emulator. While this method could substantially reduce performance overhead, how to precisely identify the set of instructions that can be perfectly emulated remains a problem.

3.3 Overview

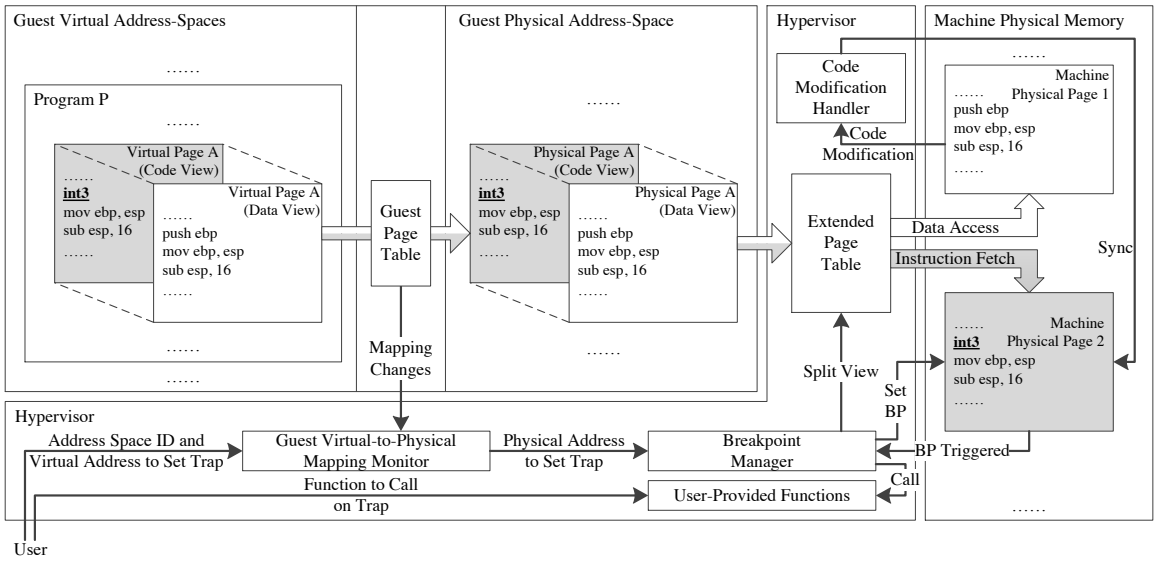


Figure 3.1.: Overview of SPIDER.

The goal of SPIDER is to provide a program debugging and instrumentation framework with flexibility, efficiency, transparency and reliability, which we define as follows:

(R1) **Flexibility:** SPIDER should be able to trap the execution of the target program at any desired instruction and data access at any memory address.

(R2) **Efficiency:** SPIDER should not introduce high performance overhead on the target program.

(R3) **Transparency:** The target program should not be able to detect the existence of SPIDER.

(R4) **Reliability:** The trap should not be bypassed or tampered with by the target program.

An overview of SPIDER is shown in Figure 4.3. For simplicity, we only show the trapping of instruction execution here. The trapping of data access using data watchpoint (Section 3.4.5) is much simpler and omitted in the figure. To trap the execution of an instruction, the user provides these inputs to SPIDER: the program address space identifier (CR3 register value in x86), the virtual address to set trap and the function to call on trap. As shown in the figure, SPIDER is mainly implemented inside the Hypervisor. The guest virtual-to-physical mapping monitor component (Section 3.4.3), which captures guest virtual-to-physical mapping changes, translates the address space identifier and the virtual address into guest physical address and invokes the breakpoint manager to set the trap. The breakpoint manager sets *invisible breakpoint* to trap the execution of the target program.

Invisible breakpoint uses the same triggering mechanism as traditional software breakpoint to inherit its flexibility (R1) and efficiency (R2). However, as discussed in Section 4.7, traditional software breakpoint is not transparent because: (1) The instructions needs to be modified in order to set breakpoint; (2) The triggering and handling of the breakpoint involves control-flow which is different from natural execution. These side-effects are neutralized in invisible breakpoint to guarantee transparency (R3). Regarding the first side-effect, the breakpoint manager uses EPT to split the code and data views (Section 3.4.1) of the guest physical page that contains the breakpoint. In the code view, which is used for instruction fetching (shown as the grey path in Figure 4.3), the instruction is modified to set breakpoint; in the data view, which is used for read/write access (shown as the white path in Figure 4.3), the

instruction is not modified at all, so the guest sees no change to the instruction. To neutralize the second side-effect, when a breakpoint is triggered, the breakpoint manager will capture the event, call the corresponding user-provided function and handle the breakpoint transparently (Section 3.4.2) so that the control-flow in the guest is the same as a natural execution. The code modification handler (Section 3.4.4) captures any modification made to the data view and synchronizes with the code view to guarantee transparency (R3); it also makes sure the breakpoint is not maliciously overwritten by the guest to guarantee reliability (R4).

3.4 Design

3.4.1 Splitting Code and Data View

SPIDER neutralizes memory side-effects of traditional software breakpoint by splitting the code and the data views of guest pages. Several existing techniques could have been used here to split the two views; however, they all have some limitations. For example, one could intercept all read accesses to modified instructions by setting the corresponding pages to not-present, and return original instructions upon read accesses. However, it would introduce significant performance overhead as every instruction fetching or data access in these pages will cause a page fault. A recent work hvmHarvard [76] tries to implement a Harvard architecture on x86 by de-synchronizing the instruction TLB (iTLB) and the data TLB (dTLB). More specifically, it tries to maintain two different virtual-to-physical page mappings in iTLB and dTLB for the code and data view respectively. To prevent the mapping of the code view from being loaded into dTLB, the page table is set to map the data view all the time; the code view is only mapped when an instruction fetching happens, and a single-step is performed in the guest to load the code view into iTLB. Unfortunately, such mechanism could not guarantee the de-synchronization of iTLB and dTLB. As the code view is readable, one could still load the code view into dTLB by executing

an instruction that reads from the page that contains it. An attacker could exploit this limitation to read from the code view and detect the modified instructions.

SPIDER splits the code and the data views of a guest *physical* page by mapping it to two host *physical* pages with *mutually exclusive* attributes. We call such guest physical page with split code and data views a *split page*. The code view of a split page is executable but not readable; the data view is readable but not executable. Both views are set to not writable to handle code modification, which will be discussed in Section 3.4.4. The mutually exclusive attributes ensure that the guest could neither read from the code view nor execute instruction from the data view of a split page. Traditionally, in x86 there is no way to set a page to *executable but not readable*; however, recent processors introduces a feature that allows one to specify such attribute in EPT entries [77]. Legacy page table still lacks such capability, which is the reason we split physical pages instead of virtual pages.

SPIDER performs on-demand transparent switching between the two views of a split page. For example, let us assume its corresponding EPT entry is currently set to mapping its code view. When a data access happens in the page, since its current view—code view is not readable, an EPT violation will occur. SPIDER will capture the event and adjust the mapping and the attribute in the EPT entry to switch to the data view. It will then resume the guest, and the data access can proceed. Switching from data view to code view works in a similar way.

It seems that SPIDER needs to switch views frequently when instruction fetching and data access in a split page are interleaved, which could result in a lot of EPT violations. However, the problem is greatly mitigated by the separation of iTLB and dTLB in x86. Given a split page, although the corresponding EPT entry could only map one of its views at any given time, the mappings of the two views can exist simultaneously in the iTLB and dTLB, respectively. For example, when SPIDER switches the page from the code view to the data view due to a data access, the mapping in the EPT is set to mapping its data view. After resuming the guest, the data access will populate the dTLB with the mapping for the data view. However,

the mapping for its code view still exists in the iTLB. Further instruction fetching will not cause any EPT violation until the mapping is evicted from iTLB.

3.4.2 Handling Breakpoints

SPIDER hides the #BP exceptions generated by invisible breakpoints and invokes breakpoint handlers at the hypervisor level to neutralize side-effects related to breakpoint handling. SPIDER sets the hypervisor to intercept all #BP exceptions generated by the guest. How to deal with intercepted #BP exceptions depends on their causes: those caused by invisible breakpoints should not be seen by the guest, while those caused by traditional software breakpoints set by the guest should be passed on to the guest transparently.

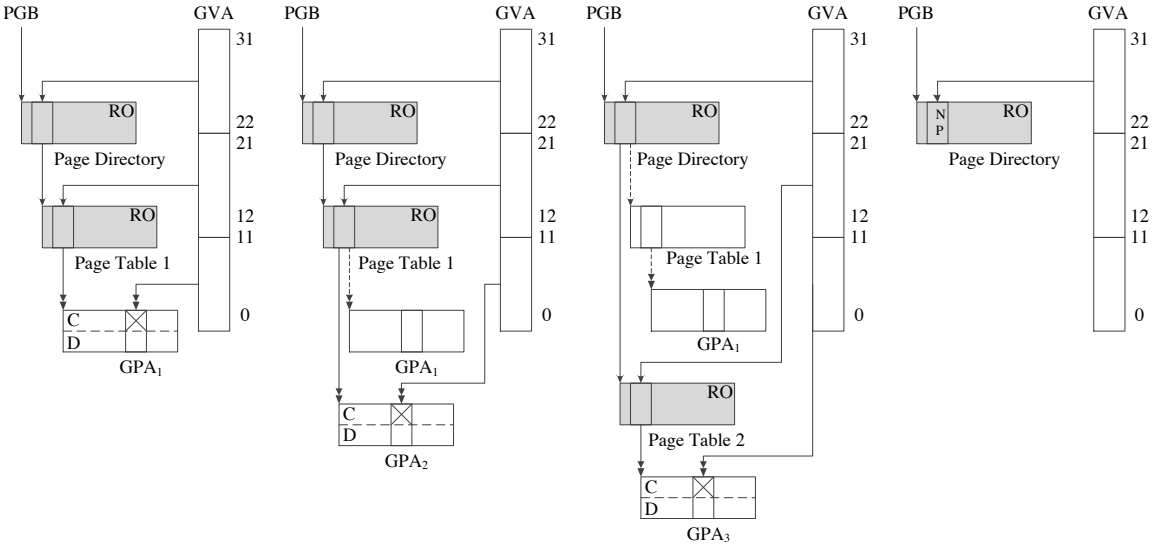
The breakpoint manager of SPIDER maintains a list which stores the *guest physical* addresses of all invisible breakpoints and their associated handlers that should be called when they are triggered. When SPIDER intercepts a #BP exception, it translates the guest instruction pointer to guest physical address by looking up the guest page table, and compares the address against the list to see whether the triggered breakpoint is an invisible breakpoint or a traditional software breakpoint. If it is a traditional breakpoint, the #BP exception will be re-injected to the guest to let the guest handle the breakpoint on its own. Otherwise, if it is an invisible breakpoint, SPIDER will call its associated handler to handle the breakpoint event. After that, SPIDER will temporarily clear the breakpoint and restore the first byte of instruction which had been replaced. Then it lets the guest single-step through the instruction.

Unlike previous work [50, 65, 76] which enables single-stepping by setting the trap flag in the guest *EFLAGS* register, SPIDER uses the monitor trap flag (MTF) which is a flag specifically designed for single-stepping in hardware virtualization. When MTF is set, the guest will trigger a VM Exit after executing each instruction. The reason why we choose not to use trap flag is that it is visible to the guest as a flag in a guest register. Despite various techniques used in previous work to hide the trap

flag, the guest could still see it. For example, if an interrupt is pending right after the guest resumes execution, the processor will invoke the corresponding interrupt handler before single-stepping through the instruction. The *EFLAGS* register is saved onto the stack, and restored after the interrupt handler returns. The interrupt handler could check the *EFLAGS* on the stack to see if the trap flag has been set. Compared with the trap flag, MTF is transparent because it could not be read by the guest. However, using MTF also causes one problem. Consider the same scenario of pending interrupt as above: when using the trap flag, the saving/restoring of the trap flag implicitly avoids single-stepping through the interrupt handler; but when using MTF, the processor will single-step through the interrupt handler before reaching the instruction. SPIDER solves this problem by "retrying": if it finds out that the guest has not executed the instruction after a single-step, it will clear MTF, set the invisible breakpoint again and resume the guest. The invisible breakpoint will be triggered again after the interrupt handler returns. This procedure repeats until the instruction is successfully executed after a single-step, and SPIDER will then clear MTF, set the invisible breakpoint again and resume the execution of the guest.

3.4.3 Monitoring Virtual-to-Physical Mapping

The invisible breakpoint provides SPIDER the ability to trap the execution of program at arbitrary *guest physical* address. However, when paging is enabled in the guest, the processor uses virtual address instead of physical address to reference memory. As paging is used by almost all modern operating systems, it is more desirable to have the ability to trap the execution of program at arbitrary *guest virtual* address in the program's address space. We define the *breakpoint address* where we want to set a breakpoint using a tuple of the address space identifier and the guest virtual address. In x86, the physical address of the base of the top-level paging structure (stored in CR3 register) serves as the address space identifier, so we write the breakpoint address as $BA = (PGB, GVA)$. If BA is mapped to a guest



Read-only paging structure is in grey color and marked with "RO". Physical page with split code and data view is represented using rectangle with a dashed line splitting the code ('C') and data ('D') view. The area in the code view with a cross inside represents the breakpoint. Arrow means path that is traversed during address translation. Double arrow indicates memory mapping. Dashed arrow or double arrow means previous path or mapping that is no longer used.

Figure 3.2.: Monitoring guest virtual-to-physical mapping.

physical address GPA, we denote it as $BA \rightarrow GPA$. If BA is not mapped to any guest physical address, we denote it as $BA \rightarrow NIL$.

To illustrate, assume the user wants to set a breakpoint at $BA_1 = (PGB_1, GVA_1)$. If $BA_1 \rightarrow GPA_1$, then we could just set an invisible breakpoint at GPA_1 to solve the problem. However, it is possible that $BA_1 \rightarrow NIL$ when we set the breakpoint (e.g., the program has not been loaded). Even if BA_1 is mapped, the mapping could change after the breakpoint is set. If the mapping changes to $BA_1 \rightarrow GPA_2$, since there is no breakpoint set at GPA_2, execution of the instruction at BA_1 will not be trapped as expected. Similarly, when BA_1 is no longer mapped to GPA_1, the breakpoint set at GPA_1 will cause problem when another address is mapped to GPA_1. Such virtual-to-physical mapping changes could happen for various reasons. For example, when the guest OS swaps out a virtual page, its corresponding physical page might be used to map another virtual page; when a write access happens in a copy-on-write virtual page, the guest OS will map it to another physical page to perform the writing; kernel-level malware could even modify the guest page table directly

to change virtual-to-physical mappings. Hence, SPIDER must monitor virtual-to-physical mapping changes to handle such scenarios correctly.

Monitoring every change of virtual-to-physical mapping requires heavy-weight techniques such as shadow page table. Fortunately, SPIDER only needs to monitor the change of virtual-to-physical mapping at each breakpoint address. In x86, the virtual-to-physical mapping is represented using multiple levels of paging structures. The number of levels depends on the operation mode of the processor, for example, whether physical address extension (PAE) or long mode is enabled. Without loss of generality, let us assume that legacy two-level paging structure is being used. As shown in Figure 3.2(a), given a breakpoint address $BA = (PGB, GVA)$, the processor traverses along a path from the page directory to the page table to translate it to a guest physical address. The only way to change the virtual-to-physical mapping at BA is to modify the paging-structure entries that is traversed during address translation, which is shown as the rectangle area in the page directory and the page table. To capture such modifications, SPIDER sets these paging structures to read-only (shown as grey) in the EPT. When there is a write access to a paging structure, an EPT violation will be triggered and captured by SPIDER. SPIDER will record the current values of paging-structure entries, then temporarily set the paging structure to writable and let the guest single-step through the instruction that performs the write access. After the single-stepping, SPIDER will read the new values of paging-structure entries and see which ones of them have been modified. After that, SPIDER will set the paging structure back to read-only to capture future modifications. The action that SPIDER performs to handle the modification depends on the type of the paging-structure entries that get modified:

Bottom-level paging-structure entries. As shown in Figure 3.2(b), when the bottom-level paging-structure entry used to translate BA is modified, the mapping changes from $BA \rightarrow GPA_1$ to $BA \rightarrow GPA_2$. As a result, SPIDER first removes the invisible breakpoint at GPA_1. Then SPIDER compares the content of the page that contains GPA_1 and the page that contains GPA_2. If they are exactly the same (which

is the case we show in the figure), then it is safe to move the breakpoint to GPA_2. Otherwise, as the code in the page has changed, it is handled in the same way as a breakpoint that might no longer be valid due to code modification (Section 3.4.4).

It is worth noting that the figure only shows the scenario where the mapping changes from a present one to another present one. The mapping might also changes from not-present to present, or oppositely. If the mapping changes from $BA \rightarrow NIL$ to $BA \rightarrow GPA$, or from $BA \rightarrow GPA$ to $BA \rightarrow NIL$, SPIDER will create/remove invisible breakpoint at GPA, respectively.

Non-bottom-level paging-structure entries. Figure 3.2(c) shows the scenario when a non-bottom-level paging-structure entry used to translate BA is modified. The virtual-to-physical mapping changes from $BA \rightarrow GPA_1$ to $BA \rightarrow GPA_3$, so SPIDER moves the breakpoint from GPA_1 to GPA_3. In addition to that, the path which the processor traverses along to perform address translation is also modified, so SPIDER also removes the read-only attribute from the paging structures in the previous path (Page Table 1) and sets the paging structures in the new path (Page Table 2) to read-only. For simplicity, we only show the change of one mapping and one path in Figure 3.2(c). In practice, modification of a non-bottom-level paging-structure entry may affect multiple paths and mappings, each of which will be handled by SPIDER individually.

There is a special case that the path used for address translation is incomplete because a non-bottom-level paging-structure entry is set to non-present, as shown in Figure 3.2(d). This could happen when setting a breakpoint at a virtual address that is not mapped in the guest, or after a non-bottom-level paging-structure entry is modified. SPIDER sets the paging structures along the path to read-only, including the one that has the non-present entry. Later, when the paging-structure entry changes from non-present to present, the path will extend, and SPIDER will set the paging structures on the extended path to read-only. After the path reaches the bottom-level paging-structure (e.g. as in Figure 3.2(a)), SPIDER could handle further modifications using standard approaches as mentioned above.

3.4.4 Handling Code Modification

When the guest tries to modify the content of a split page, the write operation will be performed on its data view. This means that if an instruction is modified, the change will not be reflected in the code view. This could lead to incorrect execution of *self-modifying programs*, and could be utilized by malware to detect the existence of SPIDER. To guarantee transparency, SPIDER must synchronize any change of the data view to the code view.

As mentioned in Section 3.4.1, SPIDER sets the data view of a split page to read-only in EPT to intercept any writing attempt. When the guest tries to write to the page, an EPT violation will be triggered and captured. SPIDER records the offset of the data OFF that is going to be written in the page. SPIDER also records the length LEN that will be synchronized by matching the instruction's op-code in a pre-built table which stores the maximum data length that could be affected by each type of instruction. Then SPIDER will temporarily set the data view to writable, and let the guest single-step through the instruction that performs the write. After that, it will copy LEN bytes from offset OFF in the data view to the same offset in the code view.

It is worth noting that the breakpoints that have been set in the page may or may not be valid after code modification. For example, if the guest overwrites an instruction with the same instruction, it indicates the guest is trying to overwrite and disable the breakpoint set at that instruction; in that case, the breakpoint is still valid and should be re-set when overwritten. But if the guest overwrites the instruction with a different instruction, re-setting breakpoint at the original place blindly may not make sense. Hence, we allow the user to specify a function which will be invoked when the page that contains the breakpoint is being modified, in which the user could perform proper actions to handle the event, such as re-setting the breakpoint at the same place, or moving it to another location after analyzing the modified code.

3.4.5 Data Watchpoint

SPIDER allows setting a data watchpoint at a specific *physical* address by adjusting the EPT entry of the guest physical page that contains the memory address to read-only (to trap write access) or execute-only (to trap both read/write access). When the page is accessed, an EPT violation will be triggered and captured by SPIDER. SPIDER will check if a watchpoint has been set on the address that is accessed in the page; if so, it will call the corresponding user-provided watchpoint handler. After that, it will temporarily set the EPT entry to writable and resume the guest to single-step through the instruction that does the memory access. When the guest returns from single-stepping, SPIDER adjusts the EPT entry again to trap future accesses. Like invisible breakpoint, data watchpoint also utilizes the virtual-to-physical mapping monitoring method (Section 3.4.3) so that it could be used to trap memory access at any *virtual* address.

3.4.6 Handling Timing Side-Effect

In hardware virtualization, since part of the CPU time is taken by hypervisor and VMEntry/VMExit, a program costs more time to run than in a native environment. Attackers could execute the RDTSC instruction to read the Time Stamp Counter (TSC) which stores the elapsed CPU cycles to detect the discrepancy. To maintain transparency, SPIDER needs to hide the CPU cycles cost by hypervisor (T_h) and VMEntry/VMExit (T_e) from the guest. SPIDER measures T_h by reading the TSC right after each VMExit and right before each VMEntry and calculating the difference. T_e is approximated by profiling a loop of RDTSC instruction in guest. SPIDER sets the *TSC-offset* field in virtual machine control structure (VMCS) to $-(T_h + T_e)$ so the value is subtracted from the TSC seen by the guest [78].

3.5 Implementation

We have implemented a prototype of SPIDER on the KVM 3.5 hypervisor. The prototype implements the design as described in Section 4.4 in the kernel module part of KVM (kvm-kmod) to provide the primitive of setting invisible breakpoint at specified virtual address in a process address space. Based on the primitive, it also implements a front-end for SPIDER in the userspace part of KVM (qemu-kvm) to provide features that make debugging and instrumentation more convenient. It is worth noting that SPIDER itself is OS-independent; However, the front-end requires knowledge of the guest OS to perform VMI [79] for some features. Currently, our front-end supports both Windows XP SP2 32-bit and Ubuntu Linux 12.04 32-bit guest. We now discuss the implementation of some features in our front-end.

Kernel Breakpoints. We have to specify an address space when setting an invisible breakpoint. For kernel breakpoints, we could specify the address space of any process as the kernel space is mapped in the same way for any process. We hence choose the address space of a long-lasting process (*init* in Linux and *System* in Windows), so the breakpoint will not be cleared due to process termination.

Monitor Process Creation. In practice, in addition to debugging running programs, it is also desirable to have the ability to get the control of a program at the moment when it is just created. For example, when analyzing malware, users often need to trap the execution at its entry point; if the malware is already running, it would be too late to set the breakpoint. To support such requirement, our front-end monitors process creation events. We set invisible breakpoints at related kernel functions to capture a newly created process and match its name against the one specified by the user. The user could get notified as soon as a process of the target program is created, and perform corresponding actions such as setting an invisible breakpoint at the entry point.

In Windows, a process is created through the *NtCreateProcessEx*[1] system call, which calls the *PspCreateProcess* kernel function to do the actual work. We set a breakpoint at the instruction right after the call to *PspCreateProcess*. When the breakpoint is triggered, we walk through the active process list at *PsActiveProcessHead* to find out the *EPROCESS* of the newly created process. The name is stored in its *ImageFileName* field.

In Linux, there are two system calls *fork* and *clone* that could be used to create a new process. They both call the same function *copy_process* to do the actual work, so we set a breakpoint at the instruction right after the call. When the breakpoint is triggered, the *task_struct* of the newly created task is in the *EAX* register as the return value. As *clone* could also be called to create thread, we need to verify the newly created task is a process by making sure its address space identifier (stored in *task_struct.mm-¿pgd*) is different from the one of the *current* task. The name is stored in the *task_struct.comm* field.

Monitor Process Termination. When a process terminates, all invisible breakpoints in its address space should be cleared. Our front-end sets invisible breakpoints at related kernel functions to monitor process termination. When a terminating process is captured, we use its address space identifier to check if it is one of our debuggee targets. If so, we will clear all invisible breakpoints in this target and remove the target.

In Windows, we set the breakpoint at the entry of the function *PspProcessDelete*, which handles cleanup when a process terminates. When the breakpoint is triggered, we read the first argument of the function from the stack, which is the *EPROCESS* structure of the process. The address space identifier is in its *Pcb.DirectoryTableBase* field.

In Linux, we set the breakpoint at the entry of the function *do_exit*, which handles the termination of the *current* task. However, the task could be a process or thread. We determine if the task is a process by checking if the *task_struct.pid* field matches

[1]Another system call *NtCreateProcess* for process creation is a wrapper of *NtCreateProcessEx*.

the *task_struct.tgid* field. The address space identifier is read from the *task_struct.mm-¿pgd* field.

The system call *execve* in Linux requires special handling. Although it does not create a new process or terminate an existing process, it changes the program running in the *current* task. We consider that both process "termination" and "creation" are involved in this procedure: the *current* task which runs the previous program is "terminated", and one that loads the new program is "created". As *execve* calls *do_execve* to do the actual work, we set a breakpoint right before the function call to capture the "terminated" *current* task, and another breakpoint right after the call to capture the "created" one.

Table 3.1.: Transparency of SPIDER and other debuggers/DBI frameworks.

Target	SPIDER	OllyDbg 1.10	IDA Pro 6.1	DynamoRIO 4.0.1-1	PIN 2.12
Software Protectors (Applied to *hostname.exe*)					
Safengine Shielden 2.1.9.0	Pass	Fail	Fail	Fail	Fail
Themida 2.1.2.0	Pass	Fail	Fail	Pass	Pass
PECompact 3.02.1 (w/ead loader)	Pass	Fail	Fail	Pass	Pass
ASProtect 1.5	Pass	Fail	Fail	Pass	Pass
RLPack 1.21	Pass	Fail	Fail	Pass	Pass
Armadillo 9.60	Pass	Fail	Fail	Fail	Pass
tElock 0.98	Pass	Fail	Fail HBP/SBP	Fail	Fail
Anti-debugging & Anti-instrumentation POC Samples					
eXait	Pass	Pass	Pass	Fail	Fail
hardware_bp.exe	Pass	Fail HBP	Fail	Pass	Pass
heapflags.exe	Pass	Fail	Fail	Pass	Pass
instruction_counting.exe	Pass	Fail HBP	Fail HBP	Fail	Fail
ntglobal.exe	Pass	Fail	Fail	Pass	Pass
peb.exe	Pass	Fail	Fail	Pass	Pass
rdtsc.exe	Pass	Fail HBP/SBP	Fail HBP/SBP	Pass	Pass
software_bp.exe	Pass	Fail SBP	Fail SBP	Pass	Pass

3.6 Evaluation

In this section, we present the evaluation of SPIDER. The experiments are done on a Thinkpad T510 laptop with Intel Core i7-3720QM 2.6GHz CPU and 8GB RAM. The host OS is Ubuntu Linux 12.10 64-bit. We use Windows XP SP2 32-bit and Ubuntu Linux 12.04 32-bit as the guest OS. We allocate 30GB image file as the hard disk and 1GB memory for the guest VM.

3.6.1 Transparency

To evaluate the transparency of SPIDER, we use two groups of Windows programs with anti-debugging and anti-instrumentation techniques. For comparison, we use SPIDER, two debuggers (OllyDbg and IDA Pro) and two DBI frameworks (DynamoRIO and PIN) to trap the execution of the target programs at certain locations. In SPIDER, the trapping is done by setting invisible breakpoints. In the debuggers, we use software or hardware breakpoints. The DBI frameworks insert instrumentations at desired instructions for trapping.

The first group of targets consists of 7 software protectors, which are widely used by both COTS software vendors and malware authors to protect their programs from being analyzed or modified. We apply these software protectors to a system program *hostname.exe* in Window XP SP2. This program reads and displays the host name of the local system; our goal is to trap the execution of its protected versions to get the host name string. We reverse-engineer the original program and find out the address of the host name string is store in the *eax* register when the program runs to the address *0x10011C6*. This also holds in the protected versions, as this program does not contain relocation information and could not be relocated by the protectors. Hence, we set the traps at *0x10011C6* in the protected versions. However, for some of the protectors, we could not set the trap when the program starts, as the instruction at *0x10011C6* is encrypted by the protectors and has not been decrypted at that time. We hence set a data watchpoint at *0x10011C6* to monitor the decryption, and set the trap once the instruction is decrypted.

We turn on all anti-debugging, anti-instrumentation and anti-VM options of the protectors when using them. The only exception is when we use Safengine Shielden, we turn off its anti-VM option. With that option on, we found that the program protected by Safengine Shielden would cease to function even when we run it in vanilla KVM *without* SPIDER; but it runs correctly in BitVisor, which is another

hardware virtualization based hypervisor. We hence conclude that the problem is due to the implementation of KVM but not SPIDER.

The second group of targets includes 8 proof-of-concept (POC) samples. Among these programs, eXait [69] aims at detecting DBI frameworks. We randomly select 10 instructions in it for trapping. The rest 7 samples implement the anti-debugging techniques commonly used in malware that is not protected by protectors, according to the statistics in [80]. Since these samples are very small (tens of instructions), we choose to trap every instruction in them.

The result is shown in Table 3.1. "Pass" indicates the program runs properly and its execution is successfully trapped at the desired location. "Fail" means the program fails to run properly in the environment even without any trap. "Fail HBP" and "Fail SBP" means the program fails to run properly after setting hardware breakpoint or software breakpoint. We can see that OllyDbg and IDA Pro fail at every target except eXait; most targets could detect their existence even when no trap is set. DynamoRIO and PIN perform better, but are still detected by 5 and 4 targets, respectively. Compared with them, SPIDER successfully maintains transparency against all 15 targets; there are 3 targets that could only be transparently trapped by SPIDER.

We also test SPIDER against techniques of detecting emulators in [71–73], which we implement as individual POC programs. We run them in SPIDER and trap every instruction in these programs as they are very short. As we expected, none of them is able to detect SPIDER, as SPIDER is built upon hardware virtualization.

3.6.2 Case Study I: Attack Provenance

In this case study, we demonstrate the use of SPIDER to improve the tamper-resistance of an existing attack provenance system BEEP [11]. Traditional attack provenance approaches are based on analysis of system event log with per-process granularity (i.e., each log entry pertains to one process). Such approaches face the

problem of dependency explosion when a long running process receives/produces a lot of inputs/outputs during its lifetime as each output is considered causally related to all preceding inputs. To solve this problem, BEEP partitions the execution of a program into individual *units*, with each unit handling an independent input request (e.g., one email or one web request) in one event-handling loop iteration. With such a finer logging granularity, BEEP is able to link each output to the truly related input(s) hence achieving higher attack provenance accuracy.

To capture the entry and exit of each unit, BEEP needs to instrument the target binary program at certain locations. BEEP uses a static binary rewriting tool PEBIL [8] to perform such instrumentation, which has several shortcomings: (1) Attackers could patch the instrumented program at runtime to disable BEEP; (2) The instrumentation needs to modify the code in the program, hence cannot be applied to programs with self-checking and self-protection mechanisms, which widely exist in COTS software to prevent malicious software manipulation. To overcome these problems, we use SPIDER to replace PEBIL for BEEP's instrumentation. The reliability of SPIDER (Section 3.3) guarantees that the instrumentation could not be circumvented or disabled. More importantly, SPIDER performs instrumentation by setting invisible breakpoints, which are transparent to the target applications.

We evaluate the effectiveness and performance of our approach using 7 Linux[2] binary programs. We first identify the instrumentation points for each program using BEEP. We then set SPIDER to monitor the creation of processes of these programs. Once a process of a target program is created, we set invisible breakpoints at the instrumentation points in its address space. The original instrumentation routines in BEEP invoke a special system call to log unit-specific events; we modify them to directly log unit events into a file in the host.

We repeated the case studies in [11] and verified the correctness of attack provenance achieved by our system. We also measure the overhead of our system over the execution of the programs in vanilla KVM. In vanilla KVM we enable Linux audit

[2]The prototype of BEEP only supports Linux currently.

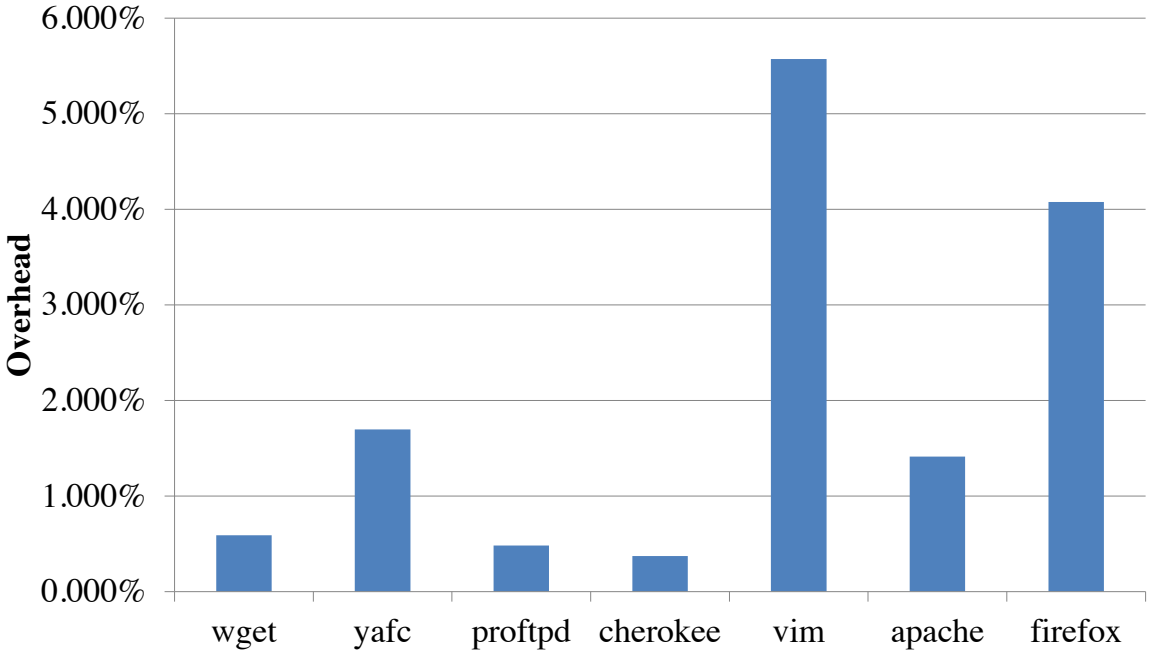


Figure 3.3.: Overhead of using SPIDER to perform instrumentation for BEEP.

system but do not perform instrumentation. For *wget* and *yafc*, we run them to download a 1.2MB file from a server 500 times. For *apache* and *cherokee*, we use the *weighttp* to generate 1 million requests with 100 threads and 100 concurrency. For *proftpd*, we use the integration test provided with it. We use the *SunSpider* benchmark for *firefox*. For *vim*, we feed it a script to replace the first character of each line with 'a' in 50000 text files. All network programs except *firefox* are evaluated in a dedicated LAN to rule out the factor of network delay. The result is shown in Figure 3.3. The overhead is less than 2% except *firefox* and *vim*. The overhead for *firefox* is slightly higher because it has more instrumentation points (24) than other programs (2∼6), which leads to more breakpoint hits. The overhead for *vim* is due to an instrumentation point which gets triggered each time the script processes a line. Users will experience much less overhead when they use *vim* interactively as the instrumentation point is triggered much less frequently.

3.6.3 Case Study II: Stealthy Introspection

We now demonstrate the use of SPIDER to reveal a possible threat to two popular Windows instant messaging programs, anonymized as IM_1 and IM_2. The threat involves the acquisition of confidential application data without user awareness. Such data usually have very short lifetime in memory and are encrypted before network transmission. Hence they are deemed difficult/impossible to acquire through memory scanning or network sniffing. We also protect the two applications using the (arguably) strongest protector Safengine Shielden, so that existing debugging/instrumentation techniques cannot be used to analyze them. Now, we show that even with those protections, confidential data could still be "stolen" by using SPIDER to trap the program at the right instruction. The stealthiness and efficiency of SPIDER make it possible to perform the attack while the programs are running normally; none of the existing techniques could achieve the same level of user-transparency and efficiency. The realism of the threat is backed by the fact that, an attacker is able to transparently hijack a running OS into a VM on malicious hypervisor (e.g., using BluePill [81]). Once that happens, SPIDER can be used to stealthily set invisible breakpoints on the target application for confidential data acquisition by the hypervisor. In the following description, such breakpoints are set on the functions and memory locations in bold font.

IM_1. We show the possibility of capturing all communication between a sender and the user. To find the function that handles messages, we search through the functions exported by the libraries of IM_1. We find a function named **SaveMsg**[3] in KernelUtil.dll and set an invisible breakpoint at the entry of that function. As expected, the function is called every time a message is received; we also find out one of its parameters is the ID of the sender. However, the message text is not directly present in the argument list, which implies that it might be part of a data structure rooted at one of the arguments. We further speculate that a message may need to

[3]Note that the binary of IM_1 does not contain symbolic information. We simply inspect the export table.

be decoded either inside **SaveMsg** or through some other related function. We find a function named **GetMsgAbstract** in the list of exported functions. The name suggests that it may need to decode a message. We set a breakpoint at its entry and another one at its return. We observe that the message text is in fact decoded as its return value. We also find out that at the entry of **GetMsgAbstract** that the value of one of its parameters is always the same as one of the parameters of **SaveMsg**, which might both point to the same opaque structure that contains the message text. Therefore, we log all messages at **GetMsgAbstract** return and associate them to individual senders by matching the parameters of **GetMsgAbstract** and **SaveMsg**. As such, we are able to identify all messages from individual senders.

IM_2**.** We show the possibility of capturing user login credentials in IM_2. We first find the functions that read the username and password. As a native Win32 application, we suspect it uses the **GetWindowTextW** Windows API function to retrieve the text from the controls in the login dialog. We set a breakpoint at the entry of that function and log all its invocations. After we rule out unrelated invocations by checking if the retrieved text matches a login credential, we find out the invocations at **0x449dbd** and **0x437a23** are for retrieving username and password, respectively. The remaining problem is to find out if the captured login credential is valid. As an error message will be displayed upon failed login, we set a breakpoint at the **MessageBoxW** function. From the call stack we could read the functions on the path of failed login. We set breakpoints on these functions too. We then do a successful login to see if it shares the same path. We find that both successful and failed logins will execute to the function at **0x48591c**, and then the path deviates. Successful login will execute to the branch of **0x485bcd**, while failed login leads to another branch. Therefore, we log the content acquired by **GetWindowTextW** when it is invoked at **0x449dbd** and **0x437a23**, and then we use the call stack path to prune those belonging to failed logins.

We verified that the confidential data (messages or login credentials) is correctly and completely acquired through stealthy introspection, without any slow-down of program execution.

3.6.4 Performance Overhead

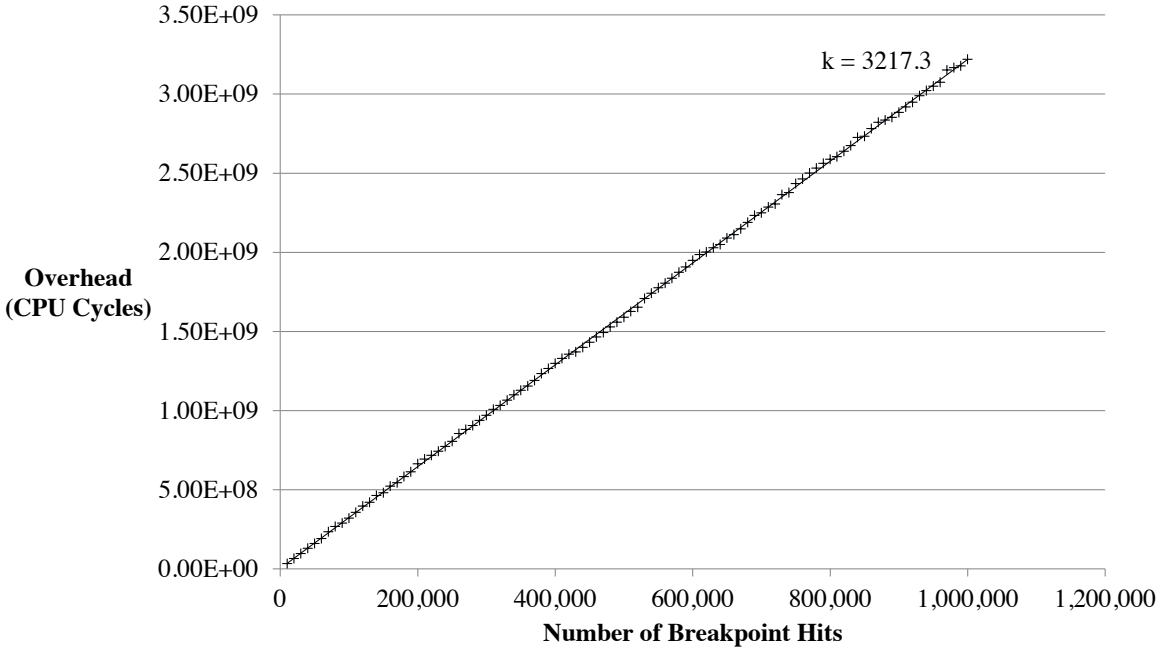


Figure 3.4.: Relation between the overhead of SPIDER and the number of breakpoint hits.

We have already presented the empirical overhead of SPIDER in our case studies in Section 3.6.2. In this experiment, we further study the overhead of SPIDER. We build a micro benchmark program that executes a loop for a given number of times. In each loop iteration, the program increments a variable 1000 times. The program executes the *RDTSC* instruction to read the CPU cycle counter before and after the loop, and calculate the difference which is the number of CPU cycles cost by the loop. We compile the program with Visual Studio 2010 in Windows.

We run the program using the parameter from 10^4 to 10^6 iterations, with a step of 10^4. The program is executed in both vanilla KVM and SPIDER; In SPIDER, we

set an invisible breakpoint at the first instruction of the loop. We obtain the number of CPU cycles cost by the loop in vanilla KVM and SPIDER, and the difference is the overhead, as shown in Figure 3.4. From the figure, we could see that the overhead is linear to the number of breakpoint hits. A single invisible breakpoint hit costs around 3217 CPU cycles. A large part of the overhead is due to the transitions between host and guest during breakpoint handling. A round-trip transition costs about 1200 cycles (measured using kvm-unit-test). This is the cost we have to pay to maximize stealthiness: To prevent any in-guest side effect, the breakpoint handler must run outside the guest VM, which means the transition is inevitable. Nevertheless, the overhead of our invisible breakpoint is still less than the breakpoint in an existing work [15] and comparable with in-guest hardware breakpoint. Considering that the cost of VMExit/VMEntry is decreasing over the years [82], the overhead of our approach is likely to be less in future processors.

We also measure the overhead of other components in SPIDER, including the cost of splitting code and data views and monitoring the guest virtual-to-physical mapping. We exclude the overhead of breakpoint hits by setting "fake" breakpoints, which use the original instruction as the breakpoint instruction instead of *int3*. The target program we use is gzip 1.2.4. We run the program in both vanilla KVM and SPIDER to compress a 98.7MB file and measure the execution time. In SPIDER, we set a breakpoint at one instruction in each page of the code section to make sure all code pages are split. The run in vanilla KVM costs 4171ms, while the run in SPIDER costs 4192ms. The overhead is less than 1% which confirms that the number of breakpoint hits is the dominant factor of overhead.

3.7 Summary

In this chapter, we present Spider, a stealthy binary program instrumentation and debugging framework. Spider uses invisible breakpoint, a novel primitive to trap execution of program at any desired instruction efficiently. Our evaluation shows

Spider is transparent against various anti-debugging and anti-instrumentation techniques. We have applied Spider in two security application scenarios, demonstrating its transparency, efficiency and flexibility.

4 IRIS: VETTING PRIVATE API ABUSE IN IOS APPLICATIONS

4.1 Introduction

Mobile devices, especially tablets and smartphones have gained tremendous popularity in recent years. Apple iOS is one of the dominating mobile platforms on the market; by the end of January 2015, Apple has sold one billion iOS devices [83]. One of its major success factors is the large number of third-party iOS applications that provide a wide variety of functionalities to users. To rapidly grow the iOS ecosystem, Apple creates the App Store which allows third-party developers to distribute their own iOS applications. As of September 2014, there are 1.3 million iOS applications available in the App Store [84].

Allowing third-party applications to run on iOS devices greatly improves the user experience. However, it also opens up the opportunity for malicious developers to attack the system and users. To prevent third-party applications from performing malicious activities, iOS employs a bunch of runtime protection mechanisms such as Sandboxing, Mandatory Access Control (MAC), Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR).

Unfortunately, even under these runtime protections, attack is still feasible through the use of private APIs. Private APIs are functions in iOS frameworks reserved only for internal uses in built-in applications. They provide accesses to various device resources (e.g. camera, bluetooth) and sensitive information (e.g. serial number, device ID), which are often not regulated by runtime mechanism. Although some resources are guarded by entitlements with MAC in recent version of iOS, there are still many that can be accessed without mediation.

As a countermeasure to the attack, Apple strictly prohibits any use of private APIs in third-party applications, according to its iOS developer license agreement [85]. To

enforce the policy, every third-party application submitted to App Store has to go through Apple's vetting process called App Review before it can be distributed to end users. Applications that pass App Review are digitally signed by Apple to prevent further modification. The signature is verified by iOS at runtime to ensure that only the original applications approved by App Review can run on iOS devices.

App Review has significantly raised the difficulty of distributing malicious applications to end users. Given the fact that very few malicious applications have been found on iOS [86], it is generally believed that App Review is quite effective. However, recent work [39, 40] shows that by constructing the names of private APIs at runtime, it is possible to invoke private APIs in third-party applications and still be able to pass the vetting process. While Apple has never publicly disclosed the technical details of App Review, these attacks clearly indicate the current vetting process is based on static analysis which is vulnerable to obfuscation. Although Apple complements automatic analysis with manual inspection [87], due to the large number of application submissions, it could only cover a small portion of all applications.

Besides Apple's App Review, there are several automated binary analysis systems [38, 88–90] proposed by security researchers to analyze iOS applications. However, these approaches also have shortcomings. System based on static analysis [38] could not resolve API names composed at runtime. Dynamic approaches [88–90] suffer from incomplete code coverage, thus would fail to detect uses of private APIs if malicious application authors place the invocations behind complicated triggering conditions.

To overcome the limitations of existing application vetting approaches on iOS, we present iRiS, an automated system that can effectively detect uses of private APIs in iOS applications. Given a binary iOS application, iRiS uses a combination of static and dynamic analysis to resolve the names of the functions being called in the program. iRiS first statically scans all function call sites and tries to resolve the names of the call targets using constant propagation and backward slicing. For the remaining call sites whose targets could not be statically determined, iRiS utilizes

dynamic binary instrumentation to drive the execution of the application to the call sites to resolve the call targets at runtime.

We have encountered and solved many challenges of performing binary analysis on iOS in the design of iRiS. Due to the closed-source nature of iOS, there is no existing dynamic binary instrumentation framework available on it. As part of our effort, we have ported Valgrind [18] to iOS and built the dynamic analysis component of iRiS on top of it. Also, most iOS applications are based on event-driven graphical user interface (GUI) which exhibits very limited behavior without human interactions. In iRiS, we propose an automated UI event handler exploration approach by using dynamic binary instrumentation to monitor the registration of event handlers and trigger them automatically.

We have used iRiS to analyze 2019 free applications on the App Store. To our surprise, the result shows that more than one hundred of these applications use private APIs. In some applications, we even identified the behavior of using private APIs to retrieve personal information (e.g. the applications installed on the device, the serial number of the device and its various components such as cameras and battery) and sending such information to advertisement providers. This clearly shows that the current application vetting approach used by Apple is insufficient to guarantee the security and privacy of iOS device users.

Our contributions are summarized as follows:

- We have ported the popular instrumentation framework Valgrind [18] to iOS. To the best of our knowledge, this is the first instruction-level dynamic binary instrumentation framework on iOS.

- We present the design and the prototype implementation of iRiS, an automated system using a combination of static and dynamic analysis to detect uses of private API in binary iOS applications.

- To show the effectiveness of our approach, we have analyzed more than 2000 iOS applications. Our result shows that a non-trivial number of iOS applications use security-critical private APIs to access and steal sensitive user information.

The rest of this chapter is organized as follows. In Section 4.2 we introduce the background. We demonstrate the practical challenges and our solutions for porting Valgrind to iOS in Section 4.3. Then we present our approach of resolving API call targets in Section 4.4. We discuss the limitations of iRiS in Section 4.6 and compare with related work in Section 4.7. Section 4.8 summarizes this chapter.

4.2 iOS Background

In this section, we introduce background about various aspects of iOS. This would help the readers to better understand our system described in later sections.

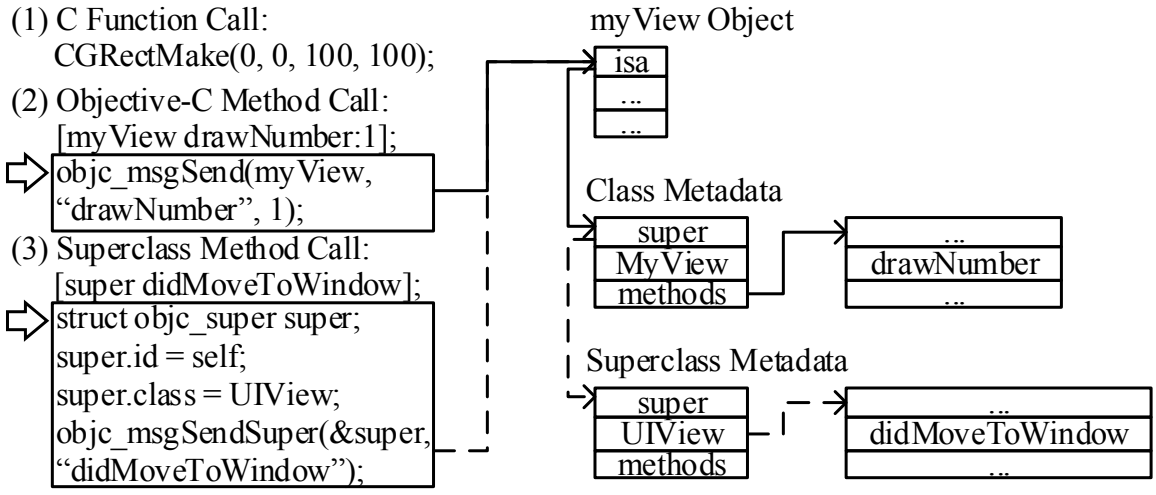


Figure 4.1.: Different forms of function invocations in iOS application.

4.2.1 Function Invocations

Objective-C is the major programming language used for building iOS applications. As an extension of the C programming language, Objective-C adds object-

oriented features such as object, class and inheritance. In Objective-C, function invocations can take several different forms as shown in Figure 4.1. Since Objective-C is a superset of C, traditional C function can be invoked as shown in case 1. In addition to that, Objective-C also supports object-oriented method calls as shown in cases 2 and 3.

The code in the boxes in cases 2 and 3 shows how Objective-C method calls are actually implemented by sending *message* to *object* through one of the `objc_msgSend` family dynamic dispatch functions. More specifically, a message is composed of a *selector* which is the literal name of the method to be invoked, and the arguments to be passed to the method. In case(2), the `drawNumber` message is sent to the `myView` object. As shown in the path along the arrows, the `objc_msgSend` dispatch function locates the metadata of the object's class `MyView`, finds the implementation (i.e. entry address) of the `drawNumber` method and then call it. Similar to other programming languages that support inheritance, if the corresponding method is not implemented in the object's class, the dynamic dispatch function searches through the object's superclasses along the class hierarchy.

Case 3 shows the use of `super` keyword to explicitly call the method in object's superclass. An `objc_super` structure containing the `myView` object and the name of its superclass `UIView` is constructed and passed to the `objc_msgSendSuper` dispatch function. The dispatch function follows the dashed path to locate the `didMoveToWindow` method in `UIView` and call it.

The dynamic features of Objective-C grant iOS developers a lot of flexibility in building their applications. Since selectors are just literal method names which contain no low-level information such as address, developers could easily construct selectors at runtime to send arbitrary messages to any object. Also, the mapping between selectors and method implementations could be modified at runtime. Such cases pose great challenges to binary analysis of iOS applications.

4.2.2 Private API

iOS provides a rich set of frameworks for building user-level applications. These frameworks are essentially directories that contain dynamic shared libraries and resources. The dynamic shared libraries expose APIs for applications in two forms: (1) as traditional C functions that are explicitly exported by the shared libraries; (2) as methods in Objective-C classes that are managed and dispatched by the Objective-C runtime.

Among all the frameworks, only some of them are public frameworks that are for uses in third-party iOS applications. The other ones, known as private frameworks, are reserved for uses in built-in applications and public frameworks only. Similar to frameworks, APIs are also categorized into public and private ones depending on whether they can be used in third-party applications. Note that public frameworks may also contain private APIs as part of their internal implementation.

Private frameworks and APIs provide many powerful functionalities that could threaten the security of the system if they are available to third-party applications. For example, the `SpringBoardServices` framework provides APIs to launch and terminate applications; the `IOKit` framework provides APIs to access mach I/O ports which could be used to obtain various device information. To prevent third-party developers from using private APIs, only public frameworks and APIs are documented and exposed by the header files in iOS software development kit (SDK). However, despite Apple's effort of concealing the prototypes of private APIs, they can still be reverse-engineered from the dynamic shared libraries in the frameworks [91].

Once their prototypes are known, calling private API functions follows the same procedure as calling public API functions. As a countermeasure, Apple requires every application submitted to the App Store to go through App Review to make sure the application binary is only linked to public frameworks and imports only public C APIs. Invocations of private Objective-C APIs are also detected because the `__objc_selrefs` section in the application binary contains all statically-known

message selectors. However, such detection is not always effective. To evade the detection, the attacker can use the `dlopen` function to load private frameworks and the `dlsym` function to locate and call private C API functions. For private Objective-C APIs, the attacker can construct the message selectors at runtime so they do not appear in the application binary.

4.2.3 iOS Runtime Security

Similar to other modern operating systems, iOS incorporates standard runtime protections such as DEP and ASLR. In addition to that, it also implements several enhanced security mechanisms as described below.

Entitlements. iOS provides fine-grained access control based on the TrustedBSD MAC framework [92]. Each application can declare a set of *entitlements* that grant specific capabilities or security permissions in iOS. The iOS kernel checks for corresponding entitlements whenever an application is trying to access guarded resources. Most entitlements in iOS are for built-in applications; the only ones available to third-party applications are for enabling iCloud service and pushing notifications. To prevent third-party developers from abusing or counterfeiting entitlements, entitlements declared in third-party applications are checked for validity during App Review and then built in to the code signatures of the application binaries. Entitlements effectively regulate the use of private APIs: without proper entitlements, even if the attacker is able to invoke the private API, iOS will refuse the attempt to access the resource. Unfortunately, there are still many resources that are not protected by entitlements in iOS.

Prohibiting dynamic code generation. iOS disallows any kind of dynamic code generation, except for applications with the `dynamic-codesigning` entitlement. This entitlement is for the built-in `MobileSafari` application to implement JIT Javascript engine and is unavailable to third-party applications. The prohibition of dynamic code generation in third-party applications has both positive and negative impact

on our system: it helps us to better disassemble third-party application binary for static analysis because there is no dynamically generated or self-modifying code; on the other hand, it also disables dynamic binary instrumentation frameworks such as Valgrind due to their needs of translating binary code at runtime. Fortunately, we can still port Valgrind to *jailbroken* iOS devices, where the kernel is patched to remove restrictions on dynamic code generation. Note this does not indicate the applications we analyze are also free to generate code at runtime; we still prohibit dynamic code generation in these applications by wrapping and checking the related system calls (e.g. `mprotect`) using Valgrind.

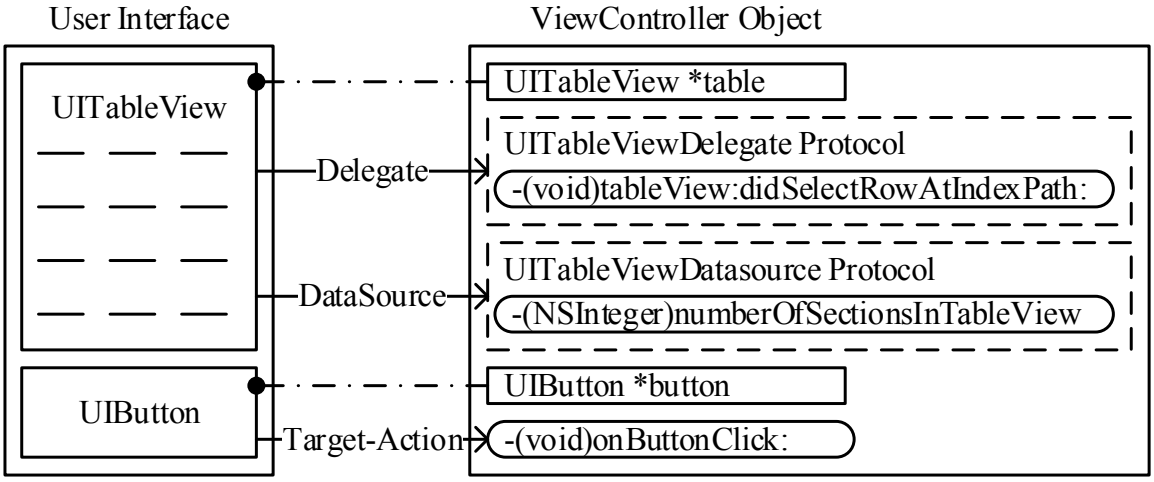


Figure 4.2.: Event driven execution of iOS application.

4.2.4 Execution of iOS Application

We use an example in Figure 4.2 to demonstrate the execution of a typical iOS application. When the application is launched, it initializes a view controller object to create and manage views. In our example, the view controller object creates a `UITableView` object and a `UIButton` object to interact with the user. To handle user inputs, it sets delegate and data source for the table view and registers a target-action

event handler to the button. These are the two design patterns for implementing event handlers in iOS, which are described as below.

Target-action. The target-action design pattern is used by all control classes (e.g. `UIButton`, `UITextField`) that are derived from the `UIControl` base class. Developers call the `addTarget:action:forControlEvents:` API to register a pair of *target* and *action* for a specific control event on a `UIControl` object. The action is the name of the Objective-C method to be invoked upon triggering the event, and the target is the object that the method is called on. In our example, the application registers the `onButtonClick:` action with the view controller object as the target, for the click event on the button. When the button is clicked, the `onButtonClick:` method will be called on the view controller object.

Delegates and data sources. *Delegates* are objects that can be assigned to a view to provide application-specific event handling logic. When an event occurs, the view sends an Objective-C message to its delegate to invoke the corresponding event handler. Usually, a delegate must conform to the protocol corresponding to the view it is assigned to, so that the view knows the required methods are indeed implemented in the delegate.

In our example, the view controller object itself is assigned to the table view as a delegate to handle events such as selecting a row in the table. When a row in the table is selected, the `tableView:didSelectRowAtIndexPath:` method will be invoked on the view controller object. The view controller object conforms to the `UITableViewDelegate` protocol which declares the event handlers for table view.

Data sources are similar to delegates except they provide application-specific data instead of logic to views. In our example, the view controller object is also assigned to the table view as a data source. When iOS renders the table view, it invokes the `numberOfSectionsInTableView` method on the view controller object to determine how many sections are there in the table.

Nib Files

Besides creating views directly in the application code, iOS application developers may also choose to load UI elements stored in Nib (NeXT Interface Builder) files. Nib files are resource files generated by Apple's UI design tool called Interface Builder, which allows developers to design UI views and related non-visual objects (e.g. view controllers) in a visualized environment. The views and objects are serialized in the format of object graph and stored in Nib files.

The UIKit framework provides several APIs to load Nib files at runtime. These Nib-loading APIs are responsible for reconstructing the views, objects and the connections among them to the same state as designed in Interface Builder. It is worth noting that in each Nib file, there is a special placeholder object called *File's Owner*. The File's Owner object is provided by the application as an argument to Nib-loading API, which serves as the link between the application code and the UI elements in Nib file. It usually contains *outlets*, which are references to the views and objects in Nib files. The outlets are connected by the Nib-loading API during the process of loading Nib Files.

To demonstrate the detail of the loading process, we still use the example in Figure 4.2, but we assume the views are loaded from a Nib file. We assume the information of the delegate, data source and the target-action event handler are all properly stored as connections to the File's Owner object in the Nib file. We provide the view controller object as the File's Owner object. According to Apple's documentation [93], the loading process consists of the following steps:

1. The Nib-loading API allocates the view objects and send them `initWithCoder:` messages to initialize the views. During the initialization of the table view, its delegate and data source are set to the File's Owner object, which is the view controller.

2. It connects the outlets (`table` and `button` variable) in the view controller to the two views by calling the `setValue:forKey:` method on the view controller. The values are the view objects and the keys are the name of the outlets.

3. It registers the `onButtonClick:` method in the view controller object as a target-action event handler to the button.

4. It sends an `awakeFromNib` message to the two views to notify them the loading is complete.

Clearly, the loading process implicitly involves invocations to many APIs, which all have to be considered in our analysis.

4.3 Porting Valgrind to iOS

In order to build the dynamic analysis component in iRiS, we ported the popular dynamic binary instrumentation framework Valgrind to iOS. Valgrind already supports ARM architecture. It also supports OS X, Apple's desktop operating system that shares the same kernel as iOS. Therefore, we could reuse the CPU-specific and OS-specific code. However, we still need to implement the parts that are specific to the combination of CPU and OS. We also encounter many practical challenges specific to iOS, some of which are discussed below. We plan to open source the ported framework to support future work on iOS security.

Calling convention of system call. Valgrind needs to interpose system calls to perform many crucial operations, such as thread and memory management. Unfortunately, the calling convention of system calls in iOS is not publicly documented. Our idea here is to infer the calling convention from the execution of the system call wrapper functions. We build a program that calls system call wrapper functions with carefully crafted arguments. Then we run the program and use GDB to set a breakpoint at those functions. Once a breakpoint is hit, we do single step until reaching a `SWI` instruction, which is used to perform system call on ARM. It is then

straightforward to infer the calling convention by observing which argument value is stored in which register or stack memory location at that point.

Reading symbols from dyld shared cache. The symbol table maintained by Valgrind is important for translating addresses to human-readable API names. Normally, Valgrind reads symbols from shared libraries when they are loaded into the address space of the application. However, there is no such loading of individual libraries in iOS. All shared libraries in iOS are combined into a single large file called dyld shared cache, which is mapped into the application's address space by the kernel when the application is loaded. To read the symbols, we invoke the `shared_region_check_np` system call to obtain the start address of the shared cache. Since the symbols of all libraries are too large to fit in the memory available to Valgrind, we read the symbols of a specific library from the shared cache only when its code is executed the first time.

Instrumenting GUI applications. In iOS, GUI applications has to be launched by sending a launch request with the bundle id of the application to `SpringBoard`. Clearly, Valgrind has to be launched this way when instrumenting GUI applications. However, the applications launched by `SpringBoard` run on behalf of the user `mobile`, which does not have the root privilege required by Valgrind. We solve this problem by setting the owner of the Valgrind executable to `root` and setting its `setuid` attribute.

4.4 Resolving API Call Targets

4.4.1 Overview

The goal of iRiS is to identify the targets of all API calls in iOS application binaries. This cannot be done with pure static analysis due to the dynamic features of Objective-C. Theoretically, dynamic analysis could resolve all the targets by utilizing approaches such as symbolic execution [94] or forced execution [95] to explore every path leading to API call. However, such approaches are infeasible in practice due to the large size of iOS applications. For example, the Facebook iOS application

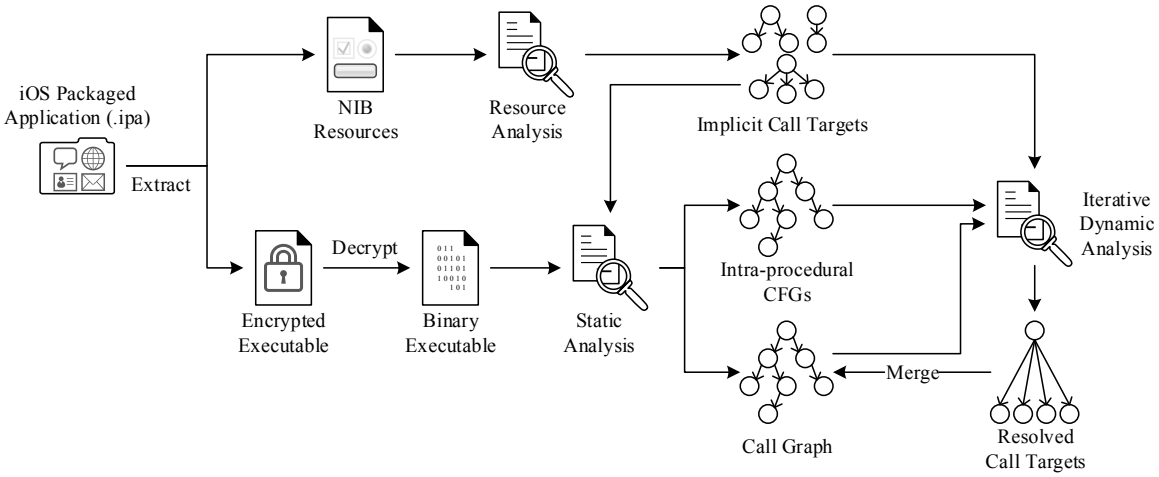


Figure 4.3.: Overview of iRiS.

binary has the size of 48MB, containing about 10 million instructions and 1.4 million branches. Binaries of such scale could not be handled by symbolic execution. Even forced execution with complexity linear to the number of branches would take several weeks to explore all the necessary paths in a single application.

To solve this problem, we adopt an approach that combines static and dynamic analysis in iRiS. Our key observation here is that the vast majority of call targets in normal iOS application binaries can be resolved using static analysis, which is fast and scales well with the size of the program. For the very few remaining call sites whose targets cannot be statically determined, we apply the slower, but more powerful dynamic analysis to get the targets from the concrete execution states at the call sites.

An overview of iRiS is shown in Figure 4.3. The input to iRiS is an iOS packaged application (with an .ipa file extension) downloaded from the App Store, which is essentially a zip file containing the application executable, resources and other meta-data. iRiS first extracts the application executable and the Nib resource files from the package. Since all applications submitted by third-party developers are encrypted by Apple before they are distributed through the App Store, iRiS needs to decrypt

the application executable to the raw binary executable before it can proceed to the analysis.

The analysis begins with resource analysis of the Nib files. For each Nib file, iRiS identifies the functions in the application binary that are implicitly invoked when the Nib file is loaded. In this way, each Nib file is represented as a set of call targets it implies. In later stages of static and dynamic analysis, upon encountering an API call that loads a Nib file, iRiS will add the call targets implied by the Nib file to the API call site.

After analyzing Nib resources, iRiS performs static analysis on the decrypted application binary executable. iRiS disassembles the binary using IDA Pro [13] and scans for all call sites. Similar to PiOS [38], iRiS tries to use backward slicing and forward constant propagation to resolve the call targets at each call site to generate an initial call graph. For each function in the binary, iRiS also generates its intra-procedural control-flow graph (CFG). The initial call graph and intra-procedural CFGs serve as guidance for the final stage of analysis.

In the final stage, iRiS iteratively resolves the remaining call sites whose targets could not be statically determined, using dynamic analysis. In each iteration, iRiS picks a call site with unresolved targets from the call graph, and uses the call graph and intra-procedural CFGs to explore paths to the call site to obtain the call targets. The resolved call targets are merged back to the call graph, which helps resolving more targets in later iterations. After all iterations are finished, the call targets in the final call graph are checked against iOS SDK headers to reveal uses of private APIs.

4.4.2 Resource Analysis

Resource analysis aims to identify application functions implicitly invoked in the process of loading a Nib file. It is infeasible to statically examine a Nib file to obtain such information since the file format is not publicly known. Our idea here is to load

the Nib file artificially using the API in the `UIKit` framework, and use Valgrind to monitor the function invocations in this process.

However, there are several challenges to load a Nib file artificially. Creating a dummy program that blindly calls the Nib-loading API would most likely fail, as a Nib file is not a self-contained entity that can be loaded in arbitrary context. For example, the objects stored in a Nib file might be of custom classes defined in the application binary. The Nib-loading API would fail when it tries to invoke the initialization methods of these objects, as they do not exist in the dummy program. Also, since the provided File's Owner object does not contain the outlets expected in the Nib file, the Nib-loading API will fail when trying to connect the outlets.

To overcome these challenges, we utilize the application itself to provide the proper context for loading the Nib files. We run the application with `DYLD_INSERT_LIBRARIES` environment variable to inject a preload shared library containing the Nib-loading code to its address space. In the preload shared library, we invoke the Nib-loading API in a function with the `constructor` attribute so it is executed before any other code (except global initialization routines) in the application binary. To handle outlets, we provide a fake File's Owner object to the Nib-loading API which ignores connections to undefined outlets by overriding the `setValue:forUndefinedKey:` method, which is the fail-safe method when the `setValue:forKey:` method for connecting outlets fails. We terminate the application by calling `exit` right after the Nib file is loaded so no unrelated code is executed.

Event handler registration functions need to be handled specially. Although the event handlers are not directly called when they are registered during Nib loading, we include them as implicit call targets so that they could be explored later in the iterative dynamic analysis stage. Since we have the concrete execution state, we can query the Objective-C runtime to get the entry addresses of the event handlers (as the parameters to the registration functions). A target-action event handler is identified if the method has the *action* selector implemented in the class of a *target* object. For delegate or data source, we enumerate the methods that are implemented in the class

of the delegate or data source object and include the ones listed in the delegate or data source protocol.

Another case that requires special handling is the function invocation to connect outlets. The `setValue:forKey:` method for connecting outlets internally invokes the setter methods of the File's Owner object to set its properties. However, since we artificially load the Nib file by providing a fake File's Owner object, the expected type of the real File's Owner object is unknown and the entry addresses of the setter methods could not be determined at this time. Therefore, we record the keys that are being set here so that the setter methods could be resolved when the class of the File's Owner object is known at later stages of analysis.

The final step of resource analysis is to prune the implicit call targets that have been obtained so far. This is because the Nib-loading API calls other functions in the `UIKit` framework or other frameworks. Such invocations might target private APIs, which is normal for internal interactions between iOS frameworks but would trigger false alarms if included in our result. We exclude the call targets that are not functions in the application by checking whether they fall in the range of the code section in the application binary.

4.4.3 Static Analysis

The goal in the static analysis stage is to build the intra-procedural CFGs and resolve call targets to construct call graphs. We build our static analysis as a plugin of the popular IDA Pro disassembler. Generating the intra-procedural CFGs is straightforward as IDA Pro already performs intra-procedural flow analysis for each function. However, the ability of IDA Pro to resolve call targets is quite limited. For traditional C function calls, IDA Pro can only identify direct call targets represented as constant relative addresses embedded in the instructions; it does not resolve indirect call targets that are stored in registers. Moreover, IDA Pro does not resolve function arguments stored in either register or stack variables. They are especially important

for analyzing the target of Objective-C method invocations. For example, even if a call to the `objc_msgSend` message dispatching function is identified, we would not be able to know the exact Objective-C method being invoked unless we resolve the message selector and the object class type from the arguments of the function.

To resolve the call targets that cannot be handled by IDA Pro, we build our analysis based on the approach proposed in PiOS [38] which consists of intra-procedural backward slicing and forward constant propagation. The basic idea is to use backward slicing to recursively identify a slice of instructions that influence the value of the register or stack variable related to the call target at the call site. Starting from the beginning of the slice, statically known constant values are propagated forwardly according to the semantic of the instructions in the slice to compute the target value.

Our static analysis consists of three passes on the application binary. Compared with the original approach in PiOS, our approach covers more forms of Objective-C message dispatching and handles implicit invocations which result in a more precise and complete call graph. The details of each pass are described as below.

Resolving C Function Calls

In the first pass, we identify all traditional C function calls and resolve their call targets. On ARM architecture, functions calls are made with `BL` (branch with link) and `BLX` (branch with link and exchange) instructions. We enumerate all these instructions in the application binary and check their operands. Constant operands representing direct call targets are already identified by IDA Pro. For register operands that contain indirect call targets, we try to use backward slicing and forward constant propagation to resolve their values. For those unresolved operands, we mark the corresponding call targets as unknown. A resolved call target is identified as an external API if the target address is one of the following two cases: (1) the address of an API in the imported symbols section or (2) the address of a *stub* function that is a trampoline for calling an external API.

Resolving Objective-C Messages

Calls to Objective-C message dispatching functions (e.g. `objc_msgSend`) are identified in the first pass. In the second pass, we try to resolve the actual Objective-C methods invoked in those message dispatching function calls.

For the message dispatching functions that invoke methods in the object's class, such as `objc_msgSend`, we use backward slicing and forward constant propagation to resolve the message selector and the object's class in the function arguments. Similar to PiOS, to resolve the object's class, we propagate not only constants, but also type information along the slice. Once the message selector and the object's class are resolved, we find the corresponding method in the class hierarchy obtained from the application using the `class-dump` [96] tool.

Other dispatching functions, such as `objc_msgSendSuper`, are used to explicitly invoke methods in object's *superclass* (Section 4.2.1). The name of the superclass is provided in an `objc_super` structure, which is pointed to by one of the function arguments. To identify the superclass, we apply two rounds of slicing and constant propagation: the first one that resolves the argument pointing to the `objc_super` structure and the second that resolves the superclass name in the structure. In most cases, the values could be successfully resolved as these functions are mainly inserted by the compiler to handle the `super` keyword in Objective-C source code, where the superclass is known at compile time.

Any Objective-C method that is not successfully resolved here is marked as unknown target to be processed later in iterative dynamic analysis.

Resolving Implicit Invocations

We resolved the targets of explicit C function calls and Objective-C method invocations in the previous two passes. In the final pass, we aim to find and resolve the targets of implicit function invocations, which are categorized and discussed as below.

Grand central dispatch. Grand central dispatch (GCD) is a runtime system to support concurrent code execution on iOS. It provides APIs (e.g. `dispatch_async`) for developers to submit functions to dispatch queues for execution. The argument of a GCD API could be a function pointer or a block object (a wrapper structure of function pointer). In either case, we apply backward slicing and forward constant propagation to get the address of the submitted function and add it as a call target.

Objective-C runtime. As we have mentioned in Section 4.2, the implementation of an Objective-C method can be changed at runtime. Therefore, it is possible for a malicious application developer to define a placeholder method, and replace its implementation with a private API function. After that, the developer could invoke the completely legitimate placeholder method to use the functionality of the private API. To prevent such attacks, we try to resolve the arguments of all functions in the Objective-C runtime that are related to retrieving or replacing the implementation of a method (e.g. `class_replaceMethod`). Although retrieving the implementation of a private API does not necessarily mean it will be called, we still consider any such behavior to be a violation due to the complexity of reasoning about method replacement statically.

Nested message passing. Some Objective-C classes provide methods to send messages, which resembles the functionality of `objc_msgSend`. For example, `NSObject`, the root class of all other Objective-C classes, provides the `performSelector` family methods which allow an object to send a message indicated by the argument to itself. The message could even be another `performSelector` which results in nested message passing. To handle such cases, we resolve the function arguments recursively until we reach the innermost message, which is added as the actual target.

Event handler registration. Similar to resource analysis, when we identify an event handler (target-action, delegate or data source) registration, we add the event handler as a call target so it could be explored later in dynamic analysis.

Nib file loading. When a call to a Nib-loading API is identified, we try to resolve the name of the loaded Nib file in the function argument. Once we know which

Nib file is loaded, we add its corresponding implicit call targets obtained in resource analysis to the call site of the Nib-loading API. We also resolve the class of the File's Owner object provided to the Nib-loading API. In resource analysis, we could not resolve the setter methods of the File's Owner object that are invoked to connect outlets, because the class of the File's Owner object is unknown at that time. With the concrete class of the File's Owner object here, those methods could be resolved now and added as implicit call targets.

The Nib-loading APIs in the `UINib` class are handled specially as they consist of two steps to load a Nib file. First the `nibWithNibName:bundle:` method is called to cache the Nib file in memory, and the Nib file is loaded at a later time using the `instantiateWithOwner:options:` method. Since it's infeasible to statically correlate the calls to these two methods, we leave them to be handled in dynamic analysis.

Algorithm 3 Call Targets Resolving Algorithm

Input:	CS - the set of unresolved call sites in static analysis
	CG - the call graph produced by static analysis
	CFG - the intra-procedural control-flow graphs produced by static analysis
Output:	CG - the updated call graph with edges to newly resolved call targets

1: $CS_n \leftarrow$ {call sites covered in the natural run}
2: $CS_{prev} \leftarrow$ {{nil} * sizeof(CS)}
3: **repeat**
4: $change \leftarrow$ {nil}
5: **for** $i \leftarrow 0$ **to** sizeof(CS) **do**
6: $CS_{rel}[i] \leftarrow$ {Nodes in paths from CS_n to $CS[i]$ in CG}
7: **if** $CS_{rel}[i] \setminus CS_{prev}[i] \neq \emptyset$ **then**
8: $targets \leftarrow$ ForceExecute($CS_{rel}[i], CS[i]$)
9: $change \leftarrow change \cup$ InsertTargets($CS[i], targets, CG$)
10: $CS_{prev}[i] \leftarrow CS_{rel}[i]$
11: **end if**
12: **end for**
13: **until** $change = \emptyset$

4.4.4 Iterative Dynamic Analysis

In the final stage of the analysis, iRiS uses dynamic analysis to resolve the call targets that cannot be determined in the static analysis stage. In dynamic analysis, as long as a function call is covered in an execution, it is straightforward to get its target and arguments from the concrete execution state at the call site. However, the task of reaching a specific call site in a dynamic execution itself is challenging. Also, we have to solve the problem of exploring the program paths that can affect the target and arguments of the function call.

We propose an iterative algorithm to find and explore the paths that could reach the target function call sites, as shown in Algorithm 3. The exploration is based on the initial call graph and the control-flow graphs of all functions generated by the static analysis. Initially (line 1), the application binary is directly executed in Valgrind without user interaction to record all call sites in the call graph that are covered in the natural run. These call sites serve as our starting points in the following rounds of exploration. The algorithm then explores the paths and updates the call graph in each iteration (line 4 to line 12). It terminates when there is no change to the call graph after an iteration (line 13).

In each iteration, we process each unresolved call site individually (line 6 to line 11). We first use depth-first search to compute the *related* call sites along the paths from the call sites covered in the natural run to the unresolved call site in the call graph (line 6). These related call sites are the ones that we use to guide the natural execution to the target unresolved call site. If the set of related call sites is different from the one in the previous iteration (line 7), the algorithm will explore paths following the new guidance to identify potential new targets at the call site (line 8).

The `ForceExecute` function (line 8) to explore paths is based on the path exploration algorithm in X-Force [95]. X-Force forces control-flow at branches to explore the basic blocks in a program. In our scenario, the call sites are analogous to the basic blocks. We denote that there is a *transition* from a call site A to another call

site B if the function F_B that contains B is one of the call targets at A. The transitions from one call site to another are analogous to the branches at the end of the basic blocks. We force those transitions to explore paths along related call sites. The application runs naturally at the start of each execution of the exploration. Once the execution reaches any related call site, we start to forcing transitions. Unlike in X-Force, where the exploration is unbounded, we limit the transitions to *related* call sites in our exploration, which ensures that each execution eventually reaches the desired unresolved call site to get its call targets. For the purpose of demonstration, let us assume the execution currently reaches the call site A, which calls the function F_B. To force a transition from A to a call site B in F_B, we force the control-flow from the entry of F_B to B by forcing branch targets in F_B. We compute the basic blocks in the paths from the entry basic block of F_B to the basic block containing B in the control-flow graph of F_B, which we denote as *safe* basic blocks since execution reaching any other basic block will not be able to reach B. In the execution starting from the entry of F_B, at each branch, we force the branch target if it does not fall in the set of the *safe* basic blocks. In this way, we guarantees the execution will reach the call site B with as few forced branches as possible.

There are some cases that need to be handled specially during the exploration, which are discussed as below:

Event handlers. In static analysis, event handlers are added as the call targets of their registration call sites. However, this is only for the purpose of path exploration algorithm; the event handler itself is not actually invoked at its registration site. Directly manipulating the call target at the registration call site to force a call to the event handler will most likely fail because it does not provide the proper context for the execution of the event handler. Therefore, the exploration of each event handler has to be handled based on its type:

- **Target-action**. A target-action event handler is registered as a pair of *action* selector and *target* object on a `UIControl` object. To trigger the event handler, we use `dispatch_async` to dispatch a call to the `sendAction:to:forEvent:`

method on the main dispatching queue of the program. When the call is dispatched, the `UIControl` object sends a message with the *action* selector to the *target* object.

- **Delegates and data sources**. We construct an `NSInvocation` to artificially invoke a specific event handler implemented by a delegate or data source. The target of the `NSInvocation` is set to the delegate or data source object, and the selector is set to the name of the event handler. The first argument is the `UIView` object which the delegate or the data source is assigned to. We pass zero to all other arguments by allocating a zeroed buffer on the stack that is as long as the size of the remaining arguments. The `NSInvocation` we construct is then dispatched on the main dispatching queue of the program.

Nib file loading with UINib. As we mentioned in Section 4.4.3, `UINib` class involves two steps to load a Nib file. We track the call to the `nibWithNibName:bundle:` method to record the name of the Nib file cached in the `UINib` object in the first step. When the program later calls `instantiateWithOwner:options:` on a `UINib` object to load the cached Nib file, we refer to the recorded information to get the name of the corresponding Nib file. At this time, we can resolve the calls involved in the Nib-loading process as both the Nib file name and the owner object are known.

Once the exploration has finished, the revealed call targets at the unresolved call site will be merged into the current call graph (line 9). Theoretically, the complexity of exploration of all possible paths is exponential to the number of related call sites. In practice, we support a number of exploration strategies (e.g. linear and quadratic) with different trade-offs between completeness and complexity. In our current implementation, we choose to use the linear complexity exploration strategy.

4.5 Evaluation

We evaluated iRiS on 2019 free applications obtained from one of the largest official App Stores. These applications are the ones listed as *popular apps* of the following

Table 4.1.: Uses of private APIs detected by iRiS in iOS applications.

Category	Framework	API Name	Functionality	#apps
Access Application Information	SpringBoardServices	SBSSpringBoardServerPort	Initialize port with SpringBoard	3
		SBSCopyApplicationDisplayIdentifiers	Obtain bundle ids of all running apps	3
		SBFrontmostApplicationDisplayIdentifier	Obtain bundle id of the front most app	3
		SBSCopyLocalizedApplicationNameForDisplayIdentifier	Get app name from its bundle id	33
	MobileCoreServices	[LSApplicationWorkspace defaultWorkspace]	Obtain the default workspace object	2
		[LSApplicationWorkspace allApplications]	Get all installed apps	1
		[LSApplicationWorkspace allInstalledApplications]	Get all installed apps	1
		[LSApplicationWorkspace applicationIsInstalled:]	Check if a specific app is installed	1
Access User's Identification Information	AppleAccount	[AADeviceInfo appleIDClientIdentifier]	Obtain the Apple ID of the device user	1
	AdSupport	[ASIdentifierManager sharedManager]	Obtain reference to the AdID manager	25
		[ASIdentifierManager advertisingIdentifier]	Obtain the device's AdID	25
		[ASIdentifierManager advertisingTrackingEnabled]	Check if advertising tracking is enabled	23
	IOKit	IOMasterPort	Initialize communication with IOKit	21
		IOServiceMatching	Find & open specified IOService object	21
		IOServiceGetMatchingService		21
		IORegistryEntryCreateCFProperty	Locate specific property (e.g. S/N)	19
		IORegistryEntryCreateCFProperties	Iterate through all properties to find information (e.g. Battery id, IMEI)	2
		IORegistryGetRootEntry		2
		IORegistryEntryGetChildIterator		2
		IOIteratorNext		2
		IORegistryEntryGetNameInPlane		2
		IOObjectRelease	Release the IOService object	2
Access User's Data/Settings	WebKit	[WebPreferences setJavaScriptEnabled:]	Enable/Disable Javascript	1
		[WebView mainFrameURL]	Get the URL of the current page	3
		[WebFrame approximateNodeAtViewportLocation:]	Get DOM Node at specified location	1
	UIKit	[UIStatusBarServer getStatusBarData]	Get precise battery level	1
		[UIView createSnapshotWithRect:]	Capture the view as an image	1
Anti-debugging	libsystem	ptrace	Prevent GDB attaching	1

categories in iTunes preview [97]: education, entertainment, finance, fitness, lifestyle, medical, productivity, social and utility. We crawled the iTunes preview website to retrieve the item ids of these applications. We download the applications through iTunes and decrypt them on iOS device using the `dumpdecrypted` [98] tool.

Among the 2019 applications, iRiS identified 149 applications that contain invocations to a total of 153 different private APIs. Among these private APIs, many of them are for implementing non-standard user interface features. For example, several applications use `setOrientation:` method in the `UIDevice` class to force the orientation of the device display. Although uses of such APIs also violate the iOS developer license agreement, we will not discuss them in detail here since they are not directly related to security. The remaining invoked private APIs that are related to security and user privacy are categorized and shown in Table 4.1, which are discussed as below.

Accessing Application Information. `SpringBoardServices` is the framework that handles application launching, management and termination on iOS. It contains

various APIs to query the status of the applications on the device. We found three applications using these APIs to obtain the bundle identifiers of the currently running and the front most application(s). After the bundle identifiers are retrieved, they are translated to application names by calling another private API. We also observed other 30 applications that call the bundle id translation API. The translation API returns a NULL pointer for non-existing bundle id, which is used by those applications to detect whether a specific application exists on the device.

We also found two applications using private APIs in `LSApplicationWorkspace` class of the `MobileCoreServices` framework to obtain the information of all applications installed on the device. The use of the `allApplications` API to get the bundle id list of all installed applications is also mentioned in a recent work [99]. We speculate that the two applications use these APIs instead of the private APIs in the `SpringBoardServices` framework because the latter ones are blocked by Apple since iOS 8.

Accessing User's Identification Information. We found one application that invokes the `appleIDClientIdentifier` API in the `AADeviceInfo` class to obtain the Apple ID of the current user. Also, there are 25 applications using the APIs in the `ASIdentifierManager` class to obtain the Advertising Identifier (AdID) of the device. AdID is an identifier which could be used to uniquely identify an iOS device. It serves as the replacement of the unique device identifier (UDID) for advertisement serving organizations after the access to UDID is disabled since iOS 7. As mentioned in Apple's documentation [100], AdID should only be accessed by advertisement serving libraries (e.g. Google AdMobs). However, we found that the `crashlytics` library, which is a library for crash reporting, calls these private APIs to access AdID in these 25 applications.

We also found 21 applications using private APIs exported by the `IOKit` framework to access various hardware information. The `IOKit` framework is for communication with low-level hardware on the iOS device. It exports various hardware components as a tree of `IOService` objects. We found that 19 of these applications

use the `IORegistryEntryCreateCFProperty` API to read for the serial number of the device from the `IOPlatformSerialNumber` property in the tree of `IOService` objects. The rest two applications use a set of private APIs in `IOKit` to iterate through the tree of `IOService` objects to find the desired information. We manually inspected these two applications and found out that they try to obtain the ID of the battery and the serial numbers of the front and back camera by looking for the properties of specific names. Further investigation reveals that the serial number of the iOS device itself is protected by entitlement since iOS 8; however, the identification information of battery and cameras are still available.

Accessing User's Data/Settings. We found a mobile browser application using private APIs in the `WebKit` framework which allows it to access data of the current web page and web browsing settings. The `WebKit` APIs are public on OS X (Apple's desktop OS); on iOS, Apple has wrapped the web browsing interface in the `UIWebView` class as a black box and make `WebKit` APIs private to prevent third-party application from accessing user's data or change web browsing settings.

We also identified two applications using private APIs in the `UIKit` framework to access sensitive user data. One of them tries to obtain the current battery level from the status bar; according to our investigation, this private API allows the application to get the precise battery level compared with using the `batteryLevel` public API in the `UIDevice` framework, which only rounds the battery level to the nearest 5%. The other application calls another private API in the `UIView` class which allows the application to capture the displayed content in a view as an image.

Anti-debugging. We found that the `Skype` application calls the `ptrace` function with the `PT_DENY_ATTACH` argument to prevent itself from being attached by GDB. Since `ptrace` is a private API that is not declared in the header files in iOS SDK, the application calls `dlsym` to dynamically retrieve the entry address and then make a call to the function.

Table 4.2.: Private API Invocations in APP_S.

Address	Private API
0xdeec2	SBSSpringBoardServerPort
0xdef46	SBSCopyApplicationDisplayIdentifiers
0xdf056	SBFrontmostApplicationDisplayIdentifier
0xfcf86	IOServiceMatching
0xfcf8e	IOServiceGetMatchingService
0xfd070	IORegistryEntryCreateCFProperty
0xfd0c2	IOObjectRelease
0xfc632	SBSCopyLocalizedApplicationNameForDisplayIdentifier
0xebfaa	[LSApplicationWorkspace defaultWorkspace]
0xebfd0	[LSApplicationWorkspace allApplications]

4.5.1 Case Study: A Suspicious Advertisement Service Provider

In this case study, we discuss our experience of identifying a suspicious advertisement service provider from the iOS applications in the App Store. Our finding started from the analysis of a utility application, anonymized as APP_S. The size of the application binary is 3.21 MB and its disassembly produced by IDA Pro contains 709894 instructions.

We used iRiS to perform thorough analysis on this application. In the static analysis stage, iRiS identified a total number of 210534 call sites (excluding the ones in API call stubs), in which 52814 were Objective-C message dispatching calls. iRiS successfully resolved most of the call targets in the static analysis; there were 21 unresolved call sites left to be examined in the iterative dynamic analysis stage. Despite the large number of statically resolved call targets, none of them actually pointed to any private API.

iRiS first performed a natural run of the application in the dynamic analysis stage. In the natural run, 13 of the 21 unresolved call sites were covered; 8 of them were targeting private APIs. The private APIs being called are the ones in the `SpringBoardServices` framework for accessing application information and the ones

in the `IOKit` framework for accessing the serial number of the device shown in Table 4.1. There were also three of them calling the APIs in the `AdSupport` framework to get the AdID of the device. However, since our later analysis shows those three calls are in an advertisement serving library, we do not considered them as private API calls.

The remaining 8 call sites not covered in the natural run were resolved iteratively with forced execution. Two of them target private APIs in the `MobileCoreService` framework for obtaining the bundle ids of all installed apps; the rest of the call targets are functions in the application binary. We closely examined the two private API call sites and found that they shared a very close ancestor on the call graph with the call sites that call private APIs in the `SpringBoardServices` frameworks. We then manually inspected the functions around the region and found out that their least common ancestor on the control-flow graph is a branch that checks if the iOS version is less than 8.0. If so, the application calls the APIs in the `SpringBoardService` framework to get the information about applications on the device; otherwise, it uses the APIs in the `MobileCoreService` framework as the former ones are blocked. Since our device runs iOS 7.0, such behavior and the additional private APIs would not be revealed, had we not used iRiS to analyze the application.

The private APIs invoked in APP_S and their call sites addresses are listed in Table 4.2. Since the application collected a lot of user privacy information, we would like to know where the information was sent to. To answer this question, we inspected the dynamic execution trace and found that there were a series of API calls right after the private API calls to post a HTTP request to the domain `http://ios.wall.youmi.net`. We then manually reverse engineered the functions along the path in the application and found out the user privacy information was encoded in the URL and sent as part of the HTTP request.

We accessed the domain at `http://www.youmi.net` which is a web site of a Chinese advertisement service provider. They provide advertisement serving library for iOS application developers to use their service, which we suspect might actually

collect the user privacy information. The library is provided as binary and headers without source code. To verify our concern, we downloaded the library, built a dummy application with it and analyzed the application using iRiS. As we expected, the application exhibited similar behavior to APP_S and sent user information to this advertisement service provider. It is worth noting that in the advertisement serving library, the Objective-C class names and method names are all obfuscated to random meaningless strings, probably to thwart the effort of manual analysis.

The suspicious advertisement service provider claims on their web site that many popular iOS applications have incorporated their advertisement serving library. In fact, in the process of analyzing more iOS applications in our pool, we did find other 20 applications that exhibited similar behavior, which indicates they also use the same library. Compared with individual iOS applications, the existence of such third-party libraries poses greater threats to user privacy as they can affect much more users by residing in a large number of applications.

4.6 Limitation

iRiS might report private API calls that do not actually happen in real executions since the application might be forced to infeasible paths during the exploration. In such case, we argue that the application should still be considered as suspicious, as it would be very unlikely that a legitimate application happens to have an infeasible path that generates a private API call.

iRiS is not able to capture private API calls in control flow generated by external input. Although dynamic code generation is prohibited in iOS, it is still possible to use return oriented programming (ROP) to introduce irregular control flow with external input, as shown in a recent work [40]. Malicious application developers might also choose to use external input, such as network data to create the message selector for Objective-C method calls. In such cases, the control flow could not be determined at the time of application vetting, thus runtime approaches such as control-flow integrity

is required to defend against the attack. Nevertheless, we consider our approach to be orthogonal to runtime defense and the two complement each other.

4.7 Related Work

The work related to iRiS can be classified into three categories: (1) dynamic binary instrumentation, (2) mobile application analysis and (3) mobile runtime hardening.

Dynamic binary instrumentation. Dynamic binary instrumentation frameworks such as PIN [17], Valgrind [18], DynamoRIO [19] and QEMU [41] are widely used for building dynamic analysis systems. All of them work on Android, but none supports iOS. Even QEMU, the full system emulator, could not run iOS since it does not emulate the required proprietary hardware of Apple. In iRiS, we ported Valgrind to iOS to build our dynamic analysis. We envision the availability of dynamic binary instrumentation on iOS would stimulate more future work on iOS security.

Mobile application analysis. There has been a lot of work in Android application analysis. Enck et al. [30] proposed TaintDroid to dynamically track privacy leaks in android applications. Lu et al. [31] presented CHEX which performs static data-flow analysis to detect component hijacking attacks. Zhang et al. [32] presented VetDroid to identify permission use behaviors in android applications using dynamic analysis. Poeplau et al. [33] applied static analysis to detect attempts of loading malicious code in Android applications. Johnson et al. [34] and Wang et al. [35] proposed to switch branch outcomes to expose hidden behavior in Android apps. However, due to the different nature of the two mobile operating systems, it is infeasible to apply these techniques on iOS. For example, most of these analysis systems are targeting the byte code running in Dalvik VM; in iOS, applications are compiled into native ARM instructions which are directly executed by the CPU. The access control in iOS is also completely different from the Android permission system.

Compared with Android, few work has been done in the domain of iOS application analysis, which is closely related to iRiS. Egele et al. [38] were the first to present PiOS,

a system to analyze privacy leaks in iOS application using static analysis. PiOS uses backward slicing and constant propagation to resolve Objective-C method calls and performs data-flow analysis to identify potential privacy leaks. In iRiS, we use similar approaches in the static analysis stage. Compared with PiOS which only handles the `objc_msgSend` message dispatching function, iRiS covers traditional C function calls, all types of Objective-C message dispatching functions and other implicit invoked functions, which result in a more complete call graph. Also, as shown in result of both PiOS and our work, static analysis itself usually is not enough to resolve all call targets in the application binary.

Szydlowski et al. [88] discussed the challenges of performing dynamic analysis on iOS applications. They proposed an approach to identify GUI views in iOS applications using image recognition. The execution of the application is driven by simulating the interaction with identified GUI views using a VNC client. Joorabchi et al. [90] proposed iCrawler to explore the UI states of iOS application by hooking into the application to inspect and exercise the UI elements. Kurtz et al. [89] proposed DiOS which utilizes UI Automation to retrieve the GUI hierarchy and interact with GUI elements. All of these three systems adopt the design of driving the execution of an iOS application by interacting with the GUI elements, which suffers from two limitations. First, it is generally infeasible to infer the interaction required to trigger a specific event handler. For example, developers might implement touch event handlers which only recognize and react to specific gestures. Second, even if proper interaction is made on the UI element, the program might refuse to transit to a new UI state when certain conditions are not met. For example, social applications usually require the user to login with his/her account at start. In such cases, the aforementioned systems would get stuck at the login screen and result in a very low code coverage. Different to the existing work, iRiS drives the execution of the application by capturing the registration of event handlers and trigger their execution programmatically, and applies forced execution so the application can get over various condition checks to reach the desired instructions.

Mobile runtime hardening. In addition to offline mobile application analysis, there also has been work focusing on hardening the execution environment of mobile applications at runtime. Davi et al. [101] proposed MoCFI to enforce control-flow integrity in mobile applications. MoCFI statically rewrites application binaries to add control-flow integrity. Following to this work, Werthmann et al. [102] proposed PSiOS which also employs static binary rewriting to add checks that enforce user-defined security and privacy policies. However, both solutions requires jailbreak of the iOS device. Recently, Bucicoiu et al. [103] proposed XiOS to prevent use of private APIs in iOS applications. XiOS statically rewrites application binaries to instrument the API call stubs and insert a reference monitor that checks for private API invocations. XiOS relies on the assumption that all calls to external APIs have to go through the call stubs. However, advanced malicious application developers could scan the address space with signatures of the target private API functions and obtain the entry addresses to call the private APIs directly, which breaks such assumption. iRiS detects uses of private APIs in the application vetting stage to complement these runtime defenses.

4.8 Summary

In this chapter, we present iRiS, an iOS application vetting system that combines static and dynamic analysis to detect uses of private APIs. Since iOS applications are usually large in size, iRiS applies static analysis to resolve the targets of most API invocations. To handle the remaining API invocations that could not be resolved statically, we propose a novel iterative dynamic analysis approach based on forced execution. We port the Valgrind dynamic binary instrumentation framework to iOS to build the dynamic analysis in iRiS. To drive the execution of the event-driven iOS applications, we propose an automated approach to trigger the execution of the event handlers. Our evaluation with over 2000 iOS applications from the official App Store shows that our technique effectively reveals many uses of private APIs that are not

detected by the official vetting process. We found a nontrivial number of applications accessing and sending out sensitive user data such as installed applications and device serial number. According to our findings, we believe that an advanced application vetting system such as iRiS is crucial for ensuring the safety of iOS device users.

5 CONCLUSIONS

Malicious software, vulnerabilities and other software security problems have become an increasing concern in recent years with the fast growth of the software industry. To guarantee the security of software users, the capabilities of analyzing and manipulating software is crucial. Binary instrumentation and transformation are essential techniques for software analysis and manipulation as binary executable is one of the most common form of software distribution. However, existing static and dynamic approaches all have serious limitations. Flexible static binary transformation is infeasible due to the lack of the capability to extract and embed binary components; malware could easily bypass analysis as dynamic binary instrumentation frameworks are not transparent to them; dynamic binary instrumentation is not even available on iOS, one of the two dominant mobile platforms. In this dissertation, we have presented three systems to improve binary instrumentation and transformation in software security scenarios.

In chapter 2 we described our static binary transformation framework Bistro which supports binary component extraction and embedding. Unlike existing detour-based or duplication-based static binary transformation approaches, Bistro supports the insertion and removal of any instruction or data at arbitrary locations in the binary. Bistro patches indirect control transfer instructions and data references with the help of inserted address translators, thus preserves the correct semantic of both the embedded component and the target program. We evaluated the performance of Bistro using the binaries in SPEC CPU 2000 benchmark and real-world software, which all shows low runtime and space overhead. We demonstrated the effectiveness of Bistro in three software security applications: (1) we carved the semantic patches for six vulnerabilities from different applications, and embedded those patches in other nine vulnerable applications to fix the same vulnerabilities; (2) we stitched

components extracted from a non-executable corpse of the Conficker worm to create a runnable sample for malware analysis and (3) we embedded functional components in kernel drivers to create trojan-ed kernel drivers which can be leveraged in defensive tasks.

In chapter 3 we discussed our dynamic binary instrumentation framework SPIDER which supports efficient and transparent trapping of the execution of binary at arbitrary instructions. SPIDER places software breakpoints and utilizes the feature in recent commodity CPUs to hide the breakpoints by splitting the instruction and data views. We also proposed an efficient mechanism of monitoring page table to accommodate updates in mappings between virtual and physical pages, which allows the user of SPIDER to set breakpoints at arbitrary virtual address in target binary. In our evaluation of SPIDER, we demonstrated that SPIDER successfully remained transparent to seven advanced software protectors equipped with state-of-the-art anti-instrumentation techniques. The performance of the invisible breakpoint placed by SPIDER is as good as traditional hardware breakpoint. We also applied SPIDER in two software security scenarios: (1) we improved existing attack provenance system by using SPIDER to instrument applications protected by advanced software protectors and (2) we used SPIDER to reveal a possible threat to two instant messenger applications protected by software protectors which allows attackers to steal user information.

In chapter 4 we described iRiS, our system for detecting uses of private APIs in iOS applications using a combination of static analysis and dynamic binary instrumentation. We ported the popular dynamic binary instrumentation framework Valgrind to the iOS mobile platform and built iRiS on top of it. iRiS applies fast static analysis to resolve most API invocations. The remaining API calls are resolved by our iterative analysis approach based on forced execution using our dynamic binary instrumentation framework. Our evaluation of iRiS prototype with over 2000 iOS applications shows that iRiS successfully identified many private API uses that were not found by Apple's official application vetting process.

5.1 Future Work

While Bistro, Spider and iRiS are effective solutions of binary instrumentation and transformation for software security applications, they all have limitations which open up the opportunities for future extensions and improvements.

Bistro currently only supports x86 instructions and Win32 PE format executables. Extending Bistro to other binary executable formats should be feasible as it does not rely on any specific feature of Win32 PE binaries. However, extending Bistro to other instruction sets (e.g. ARM) requires us to throughly enumerate and handle the control transfer instructions and memory addressing modes. Another possible extension to Bistro is to enable binary transformation at load time or even runtime. The address translator snippets inserted by Bistro are able to handle the address changes caused by transformation; however, the address mapping needs to be updated each time a transformation happens, which calls for a new data structure that can be more efficiently modified rather than perfect hashing to store the mapping.

As we have discussed in chapter 3, Spider executes all instrumentation functions in the context of the hypervisor to prevent potential in-guest side effects. Compared with Spider, in most other dynamic binary instrumentation engines, the instrumentation routine is usually executed in the context of the instrumented program itself. Although such difference does not affect the functionality for passive instrumentation, it does make writing an instrumentation tool unnecessarily verbose and complicated. An extension to Spider to handle this problem would be a dynamic execution context translator which automatically translates the guest execution context to the context of the hypervisor for the instrumentation routines. In this way, instrumentation tool authors could write their tools as if they run in the guest, and even existing instrumentation tools for other frameworks could be ported to Spider with slight modification.

Although we handled the timing side-effect by manipulating the Time Stamp Counter in SPIDER, it might still be detected with timing attacks using external time sources (e.g. NTP servers, wall clocks). Also, performance counters such as number of retired instructions might reveal the existence of SPIDER due to the execution of additional instructions. An attacker might also find the footprint of SPIDER by probing the Translation Look-aside Buffer (TLB) or various caches. We plan to study the approaches of countering such side-channel attacks to SPIDER in the future.

SPIDER is currently only implemented in x86 architecture. However, our key idea of splitting the code and data view is not specific to the x86 architecture. We plan to port SPIDER to ARM when hardware virtualization support on ARM is mature.

The list of private APIs identified by iRiS might be incomplete since iRiS cannot afford to explore all paths leading to the API call sites in dynamic analysis. In our current implementation, we adopt the linear exploration strategy, which does not reveal private API calls that require a combination of multiple functions to trigger. A possible future work to alleviate such problem should experiment and evaluate more complex exploration strategies, such as quadratic search to achieve better path coverage. One possible optimization to reduce the search space is to apply taint analysis and only explore the branches whose predicates are tainted by the input. Another direction for reducing the total time of application vetting is to parallelize the call targets resolving algorithm so multiple devices could be used to speed up the analysis of one application.

The current implementation of iRiS does not cover all types of implicit function invocations in iOS frameworks. For example, the `NSTimer` class allows developers to register a callback function which is called when the timer fires. Handling all such implicit function invocations is an extension to effectively improve the completeness of iRiS, which requires careful examination of all classes and APIs provided in iOS frameworks.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-Time Binary Rewriting Techniques for Program Compaction. *ACM Transactions on Programming Languages and Systems*, 27(5):882–945, 2005.

[2] R. Muth, S.K. Debray, S. Watterson, and K. De Bosschere. alto: A Link-Time Optimizer for the Compaq Alpha. *Software: Practice and Experience*, 31(1):67–101, 2001.

[3] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[4] Alan Eustace and Amitabh Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the USENIX 1995 Technical Conference*, pages 25–25, 1995.

[5] Bryan Buck and Jeffrey K Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[6] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.

[7] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium*, volume 3, pages 14–14, 1999.

[8] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 175–183. IEEE, 2010.

[9] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 154–172. 2011.

[10] Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[11] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Network and Distributed Systems Security Symposium*, 2013.

[12] GDB. http://www.gnu.org/software/gdb/.

[13] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com/idapro/.

[14] Oleh Yuschuk. Ollydbg. http://www.ollydbg.de/.

[15] Amit Vasudevan and Ramesh Yerraballi. Stealth Breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 383–392, 2005.

[16] Amit Vasudevan. Re-Inforced Stealth Breakpoints. In *Proceedings of the 2nd International Conference on Risks and Security of Internet and Systems*, pages 59–66, 2009.

[17] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200, 2005.

[18] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. volume 42, pages 89–100, 2007.

[19] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[20] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 36–47, 2003.

[21] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163, 2006.

[22] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 135–146, 2012.

[23] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. In *ACM SIGPLAN Notices*, volume 47, pages 133–144, 2012.

[24] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-Grained Malware Analysis using Stealth Localized-Executions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[25] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th Annual EICAR Conference*, 2006.

[26] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, pages 1–25. 2008.

[27] Prashanth P Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 137–147, 2007.

[28] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating Emulation-Resistant Malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, pages 11–22, 2009.

[29] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. *ACM SIGPLAN Notices*, 47(7):227–238, 2012.

[30] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[31] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

[32] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 611–622, 2013.

[33] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[34] Ryan Johnson and Angelos Stavrou. Forced-Path Execution for Android Applications on x86 Platforms. In *Proceedings of the 7th IEEE International Conference on Software Security and Reliability-Companion*, pages 188–197, 2013.

[35] Zhaohui Wang, Ryan Johnson, Rahul Murmuria, and Angelos Stavrou. Exposing Security Risks for Commercial Mobile Devices. In *Computer Network Security*, pages 3–21. 2012.

[36] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, pages 5–8, 2012.

[37] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, volume 12, 2012.

[38] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[39] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. Launching Generic Attacks on iOS with Approved Third-Party Applications. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 272–289, 2013.

[40] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[41] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[42] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 29–44, 2010.

[43] J. Caballero, N.M. Johnson, S. Mccamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[44] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reuse-Oriented Camouflaging Trojan: Vulnerability Detection and Attack Construction. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[45] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. *SRI International*, 2009.

[46] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1):4, 2009.

[47] N.M. Johnson, J. Caballero, K.Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 347–362, 2011.

[48] Halvar Flake. Structural comparison of executable objects. In *Proceedings of the 1st SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2004.

[49] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, 2006.

[50] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[51] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, pages 2732–2733, 2004.

[52] A. Slowinska, T. Stancescu, and H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[53] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[54] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[55] D.C. Schmidt. GPERF: A Perfect Hash Function Generator. *More C++ Gems*, pages 461–491, 2000.

[56] N. Falliere, L.O. Murchu, and E. Chien. W32. Stuxnet Dossier. *White paper, Symantec Corp., Security Response*, 5, 2011.

[57] Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 299–308, 2012.

[58] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216, 1994.

[59] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, page 15, 2006.

[60] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 75–88, 2006.

[61] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23th Annual Computer Security Applications Conference*, pages 421–430, 2007.

[62] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary Obfuscation using Signals. In *Proceedings of the 16th USENIX Security Symposium*, pages 275–290, 2007.

[63] M. Prasad and T. Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[64] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pages 98–115, 2008.

[65] Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proceedings of the 25th Annual Computer Security Applications Conference*, pages 441–450, 2009.

[66] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 189–198, 2012.

[67] Sebastian Vogl and Claudia Eckert. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. In *Proceedings of the European Workshop on System Security*, volume 12, 2012.

[68] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[69] F Falcon and N Riva. Dynamic Binary Instrumentation Frameworks: I Know You're There Spying on Me. In *Reverse Engineering Conference*, 2012.

[70] Kevin P Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996.

[71] Peter Ferrie. Attacks on Virtual Machine Emulators. *Symantec Advanced Threat Research*, 2006.

[72] Peter Ferrie. Attacks on More Virtual Machine Emulators. *Symantec Technology Exchange*, 2007.

[73] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting System Emulators. In *Information Security*, pages 1–18. 2007.

[74] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[75] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[76] Michael Grace, Zhi Wang, Deepa Srinivasan, Jinku Li, Xuxian Jiang, Zhenkai Liang, and Siarhei Liakh. Transparent Protection of Commodity OS Kernels Using Hardware Virtualization. In *Security and Privacy in Communication Networks*, pages 162–180. 2010.

[77] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3C.

[78] Zhui Deng, Dongyan Xu, Xiangyu Zhang, and Xuxiang Jiang. IntroLib: Efficient and Transparent Library Call Introspection for Malware Forensics. *Digital Investigation*, 9:S13–S23, 2012.

[79] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[80] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Proceedings of the Black Hat Technical Security Conference*, 2012.

[81] Joanna Rutkowska. Subverting Vista Kernel for Fun and Profit. *Black Hat Briefings*, 2006.

[82] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the USENIX Annual Technical Conference*, volume 12, pages 35–35, 2011.

[83] BusinessInsider. Apple Has Shipped 1 Billion iOS Devices. http://www.businessinsider.com/apple-ships-one-billion-ios-devices-2015-1.

[84] Statista. Number of Available Apps in the Apple App Store. http://www.statista.com/statistics/263795.

[85] Apple. iOS Developer Program License Agreement. https://developer.apple.com/programs/terms/ios/standard/ios_program_standard_agreement_20140909.pdf.

[86] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011.

[87] 9to5mac. Former Apple Employee Discusses the App Store Review Process. http://9to5mac.com/2012/07/04/former-apple-employee-discusses/.

[88] Martin Szydlowski, Manuel Egele, Christopher Kruegel, and Giovanni Vigna. Challenges for Dynamic Analysis of iOS Applications. In *Open Problems in Network Security*, pages 65–77. 2012.

[89] Andreas Kurtz, Andreas Weinlein, Christoph Settgast, and Felix Freiling. DiOS: Dynamic Privacy Analysis of iOS Applications. Technical Report CS-2014-03, Department of Computer Science, Friedrich-Alexander-Universitt Erlangen-Nrnberg, 2014.

[90] Mona Erfani Joorabchi and Ali Mesbah. Reverse Engineering iOS Mobile Applications. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 177–186, 2012.

[91] Nicolas Seriot. iOS Objective-C Runtime Headers. https://github.com/nst/iOS-Runtime-Headers.

[92] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In *Proceedings of the USENIX Annual Technical Conference*, pages 285–296, 2003.

[93] Apple. Nib Files. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html.

[94] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[95] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[96] Steve Nygard. Class-Dump. http://stevenygard.com/projects/class-dump/.

[97] Apple. iTunes Preview. https://itunes.apple.com/cn/genre/ios/id36?mt=8.

[98] S. Esser. dumpdecrypted. https://github.com/stefanesser/dumpdecrypted.

[99] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John CS Lui. Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 463–474, 2015.

[100] Apple. ASIdentifierManager Class Reference. https://developer.apple.com/library/ios/documentation/AdSupport/Reference/ASIdentifierManager_Ref/.

[101] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Network and Distributed Systems Security Symposium*, 2012.

[102] Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. PSiOS: Bring Your Own Privacy & Security to iOS Devices. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 13–24, 2013.

[103] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. XiOS: Extended Application Sandboxing on iOS. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 43–54, 2015.

VITA

VITA

Zhui Deng received his B.E. degree in computer science and technology from Tsinghua University in 2007 and M.S. degree in computer science and technology from Tsinghua University in 2010. He then obtained his Ph.D. degree in computer science from Purdue University in 2015 under the direction of Professor Dongyan Xu and Professor Xiangyu Zhang. His research interests are in the areas of virtualization, software security and binary program analysis. He interned with Google Inc. in 2014 during his Ph.D. studies and joined as a full-time software engineer in the fall of 2015.