System Techniques for Reverse Engineering Mobile Applications

by

Yibin Liao

(Under the Direction of Kang Li)

Abstract

Reversing mobile application has become a complicated and time-consuming task since various anti-reverse engineering techniques (e.g., packing, anti-debugging, anti-emulator, obfuscation, etc.) employed by latest mobile applications make current reverse engineering techniques ineffective. Many approaches have been used, such as machine learning, dynamic instrumentation, etc. However, little has been done from a systems perspective to provide effective, robust and efficient solutions. The arms race between reverse engineering and anti-reverse engineering has brought new challenges to the design of modern mobile security analysis.

This dissertation focuses on the systems aspect of the challenges that reverse engineering researchers face in designing various reversing approaches. Designing a system that collecting, organizing, and evaluating facts about a mobile application and the environment in which it operates is an effective way for automating reverse engineering analysis and fight against anti-reverse engineering techniques on mobile platforms.

We designed a *virtual machine instrumentation system*, an automatic analysis platform that provides a comprehensive view of packed Android applications behavior by conducting multi-level monitoring and information flow tracking. This system is capable of identifying

packed Android applications, extracting hidden code during the execution and performing unpacking process for packed Android Applications.

We designed *MobileFindr*, an on-device trace-based function similarity identification system for iOS platform. *MobileFindr* runs on real mobile devices and mitigates many prevalent anti-reversing techniques by extracting function execution behaviors via dynamic instrumentation, then characterizing functions with collected behaviors and performing function matching via distance calculation. We have evaluated *MobileFindr* using real-world top-ranked mobile frameworks and applications. The experimental results showed that *MobileFindr* outperforms existing state-of-the-art tools in terms of better obfuscation resilience and accuracy.

INDEX WORDS: Reverse engineering, Instrumentation systems, Dynamic analysis, Trace-based, Function similarity, Mobile applications

System Techniques for Reverse Engineering Mobile Applications

by

Yibin Liao

B.E., Nanchang Hangkong University, China, 2008

M.S., University of Louisiana at Lafayette, USA, 2011

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2018

System Techniques for Reverse Engineering Mobile Applications

by

Yibin Liao

Approved:

Major Professor: Kang Li

Committee: Maria Hybinette
Kyu Hyung Lee

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2018

Acknowledgments

I would like to thank Dr. Hybinette and Dr. Lee for all their help and guidance. I would also like to thank my beloved wife, Sisi Ye and my family for their continual support and encouragement through my PhD program.

To my advisor, Dr. Kang Li, I insist that this is the best work that i can accomplish. Addition time and effort in graduate school can neither improve my dissertation presentation nor enhance my research contributions.

Table of Contents

Page

Acknowledgments iv

List of Figures vii

List of Tables viii

Chapter

1 Introduction 1

1.1 Background 1

1.2 Research Problems 2

1.3 Dissertation Contributions and Roadmaps 3

2 Literature Review 5

2.1 Reverse Engineering 5

2.2 Anti-reverse Engineering Techniques 7

2.3 Software Instrumentation 13

3 System Instrumentation for Analyzing Packed Android Applications 20

3.1 Introduction 20

3.2 Background 22

3.3 System Design and Implementation 28

3.4 Evaluation 32

3.5 Discussion and Future Work 34

3.6 Conclusion 37

4 MobileFindr: Function Similarity Identification for Reversing Mobile Binaries . . . 45
4.1 Introduction . . . 45
4.2 Background . . . 48
4.3 Design and Implementation . . . 52
4.4 Evaluation . . . 60
4.5 Discussion . . . 67
4.6 Related Work . . . 68
4.7 Conclusion . . . 70
5 Summary . . . 72
Bibliography . . . 74
Appendix
A Android System Instrumentation Configuration . . . 82
B MobileFindr Usage . . . 84

List of Figures

2.1 Process Status 8
2.2 Self-attached Anti-debugging 9
2.3 Decompiled Code Tree Comparison between Unpacked and Packed Version . 10
2.4 Original Control Flow Graph 17
2.5 Control Flow Graph After Flattening 18
2.6 Bogus Control Flow 19
3.1 How Android Applications are Built and Run 39
3.2 JNI General Work Flow 40
3.3 Decompiled Code for Unpacked Android Applications 41
3.4 Decompiled Code for Packed Android Applications 41
3.5 Comparison of Unpacked and Packed Application for Dumping DEX Code Content from Memory 42
3.6 Code Obfuscation for Native Methods 42
3.7 An Overview of Automatic Detecting and Unpacking for Packed Android Applications 43
3.8 Android Runtime Class Loading 44
4.1 A Motivating Example: Code 51
4.2 A Motivating Example: CFG 51
4.3 Schematic Overview of Trace-based Function Similarity Mapping System . . 52
4.4 Partial vs Full ASLR in iOS 54
4.5 Function Mapping between Obfuscated Version and Non-obfuscated Version 61
4.6 Function Mapping Evaluation for Popular Third-party Frameworks 66
4.7 Function Mapping Evaluation in Real-world Applications 67

List of Tables

2.1 Instruction Replacing 13
3.1 Complete Instrumented Functions in DVM and ART 32
3.2 Building Packed Android Applications 33
3.3 Detecting Packed Android Applications in Real World 34
4.1 Different Obfuscation Types and Flag Settings 63

Chapter 1

Introduction

1.1 Background

Reverse engineering is the process of taking a software program's binary and recreating it so as to trace it back to its original source code. Reverse engineering has been widely used in computer hardware and software for many purposes: to enhance product features without knowing the source; for testing code compatibility; for finding security flaws; as a way to understand the design of malicious code.

However, reverse engineering mobile applications is a complicated and time consuming task since different mobile platforms and frameworks require specific domain knowledge and a significant amount of manual effort. In addition, mobile application frequent updates lead to a lot of repeated work in reverse engineering. What's worse, various anti-reverse engineering techniques (e.g., packing, anti-debugging, obfuscation, etc.) employed by latest mobile applications further make current reverse engineering techniques ineffective. For instance, Android packers usually adopt complex code hiding techniques to hide original executable files in Android applications. This packing service has been widely used by a huge number of Android malware. In addition, code obfuscation as well as anti-debugging and anti-emulator techniques employed by mobile applications make reverse engineering ineffective when identifying binary code at function level.

There is a number of approaches have been proposed. However, little has been done from a systems perspective to provide effective, robust and efficient solutions. The arms race between reverse engineering and anti-reverse engineering has brought new challenges to the

design of modern mobile security analysis. Therefore, an automated, systematic method is necessary to meet the demand of reverse engineering mobile applications.

1.2 Research Problems

Android and iOS dominated smart-phone market with a share of 96.79% until 2018 [1]. In this dissertation, we focus on the problems and challenges on both Android and iOS platforms.

1. Android packing service become more and more popular recently. There is a number of unpacking approaches have been proposed. Our questions here are:
   (a) Do current unpacking approaches work effectively for recent Android packers?
   (b) How can we achieve a more comprehensive understanding of current packers?
2. Unlike Android, which is an open source operating system, iOS is a close-source platform. Nowadays iOS developers heavily rely on code obfuscation to evade code detection. However, identifying binary code at function level has been applied to a broad range of software security applications and reverse engineering tasks. The questions here are:
   (a) Do current function identification approaches work for iOS platforms?
   (b) What features are useful for function identification?
   (c) How these features are captured on iOS platforms?
   (d) How to characterize a function with such features?

1.3 Dissertation Contributions and Road-maps

In this dissertation, we tackle the problem above by introducing advanced and novel system techniques. We propose various state-of-the-art system techniques to fight against the anti-reverse engineering, deploy our systems in real-world environments, and show the effectiveness and advantages of our approach. Particularly, we make the following contributions as answers for the questions above.

- We conducted analyses of known Android packers as well as unpacking approaches appeared in recent years and summarize the techniques used by them and the limitations of current unpacking approaches in this dissertation.

- We proposed a multi-level virtual machine instrumentation system with the capability to automatically identify packed Android applications from unpacked one, reassemble the hidden executable files and provide a comprehensive view of packing behaviors.

- We addressed the limitations of current function identifying approaches, and proposed a novel approach, *trace-based function similarity mapping*, to perform function similarity measurement on iOS platforms. Our approach exhibits stronger resilience to various anti-reverse engineering techniques for iOS applications. To best of our knowledge, this is the first work having such ability on iOS platforms.

- We have proposed a variety of dynamic features to record during the function execution, which allow us to approximate the semantics of a function without relying on the source code access.

- We have demonstrated the viability of our approach for top-ranked real-world mobile frameworks and applications.

The remainder of this dissertation is organized as follows. After Literature Review in Chapter 2, we introduce our Android multi-level instrumentation system in Chapter 3. We

first discuss background of Android packer, and then present detailed design and implementation of this instrumentation system, followed by evaluation with open source and real world Android applications. In Chapter 4, we present the work of MobileFindr. We introduce the background of function similarity mapping as well as challenges first. Then we presented details of our system design and implementation. After that we present our evaluation and results, discuss the limitations and related work. Chapter 5 summarize the dissertation. Appendix A shows configuration settings that trigger the unpacking process of our virtual machine instrumentation system. Appendix B shows how to deploy and use MobileFindr to perform function similarity mapping.

Chapter 2

Literature Review

This section presents the state-of-the-art in reverse engineering. The first part introduces the background of reverse engineering, presents popular techniques and tools that help for reverse engineering mobile applications, including various debuggers, disassemblers, decompilers, etc. The second part of the literature describes various anti-reverse engineering techniques deployed by mobile applications. The third part introduces software instrumentation, which has been used in this thesis work.

2.1 Reverse Engineering

Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object. This practice is now frequently used on computer hardware and software. Software reverse engineering involves reversing a program's machine code (the string of 0s and 1s that are sent to the logic processor) back into the source code that it was written in, using program language statements.

2.1.1 Static vs Dynamic

To understand a software program, both static and dynamic information are useful for reverse engineering. Static information describes the structure of the software as it is written in the source code, while dynamic information describes the runtime behavior. Static analysis involves analyzing a programs source code or machine code without running it. In reverse engineering, many tools perform static analysis, in particular decompilers and disassembler. Dynamic analysis involves analyzing a client program as it executes. Many tools perform

dynamic analysis, for example, profilers, checkers and execution visualizers. Tools performing dynamic analysis must instrument the client program with analysis code.

Both static and dynamic analysis result in information about the software artifacts and their relations. The dynamic analysis also produces sequential event trace information, information about concurrent behavior, code coverage, memory management, etc. Although there are benefits for conducting static and dynamic analysis as separate tasks, an analyst can realize the value provided by conducting both techniques when reverse engineering complex mobile applications.

2.1.2 Reverse Engineering Tools for Mobile Applications

There are numerous reversing tools available for mobile applications such as Apktool, baksmali, dex2jar, jd-gui and IDA Pro. Apktool [2] and baksmali [3] are free tools for reverse engineering Android applications. It can convert Android executable (DEX) to human-readable Dalvik byte-code. Android developer can use Dex2jar [4] to convert DEX file to Java class file, and then open it in JD-GUI [5] to display Java source code. The most powerful commercial disassembler is IDA Pro [6], published by Hex-Rays. It can handle binary code for a huge number of processors and has open architecture that allows developers to write add-on analytic modules. Hex-Rays Decompiler [7] is a IDA Pro extension that converts native processor code into human readable C-like pseudo-code text.

Debugger helps developer to understand how the program behaves at runtime without modifying the code, and allows the user to view and change the running state of a program. With the release of Xcode 5, the LLDB debugger [8], which is part of the LLVM compiler development suite, becomes the foundation for the debugging experience on Apple platforms. LLDB is fully integrated with Xcode and provides deep capabilities in a user-friendly environment. For Android platform, both LLDB and JDB (Java debugger) are integrated in the Android Studio debugger [9]. By default, Android Studio automatically choose the best option for the code you are debugging. For example, if you have any C or C++ code in

the project, Android studio debugger select LLDB to debug your code. Otherwise, Android Studio uses the Java debug type.

2.2 Anti-reverse Engineering Techniques

The software security community relies on such reverse engineering tools to analyze and validate programs. However, various anti-reverse engineering techniques employed by the latest mobile applications further make current reverse engineering tools ineffective. We summarize the common anti-reverse engineering techniques as the following:

2.2.1 Anti-debugging

Android is based on Linux kernel. In Linux, one process can attach to another process for debugging. Developers can insert anti-debugging code stubs (e.g., attach to themselves using ptrace) to interfere dynamic analysis based tools (e.g., gdb). In other words, if an application (target process) attaches to itself at runtime, gdb cannot attach to it, thus further debugging operations are prohibited.

For instance, Figure 2.1 shows the running process status of an Android Application called *com.csair.mbp*. The process ID for this application is 26883. The highlighted *TracerPid* indicates that there is another process (ID: 26968) attached to this process. Figure 2.2 lists all running processes. The first column is the USER ID. The second is the PID, and the third is the PPID, which is the parent PID. Therefore, both process 26883 and 26968 are from same application. Process 26883 launches child process 26968, which then attached itself to avoid any other processes to attach for debugging. Some of the mobile applications will also check whether special threads, such as JDWP (Java Debug Wire Protocol) thread, have been attached.

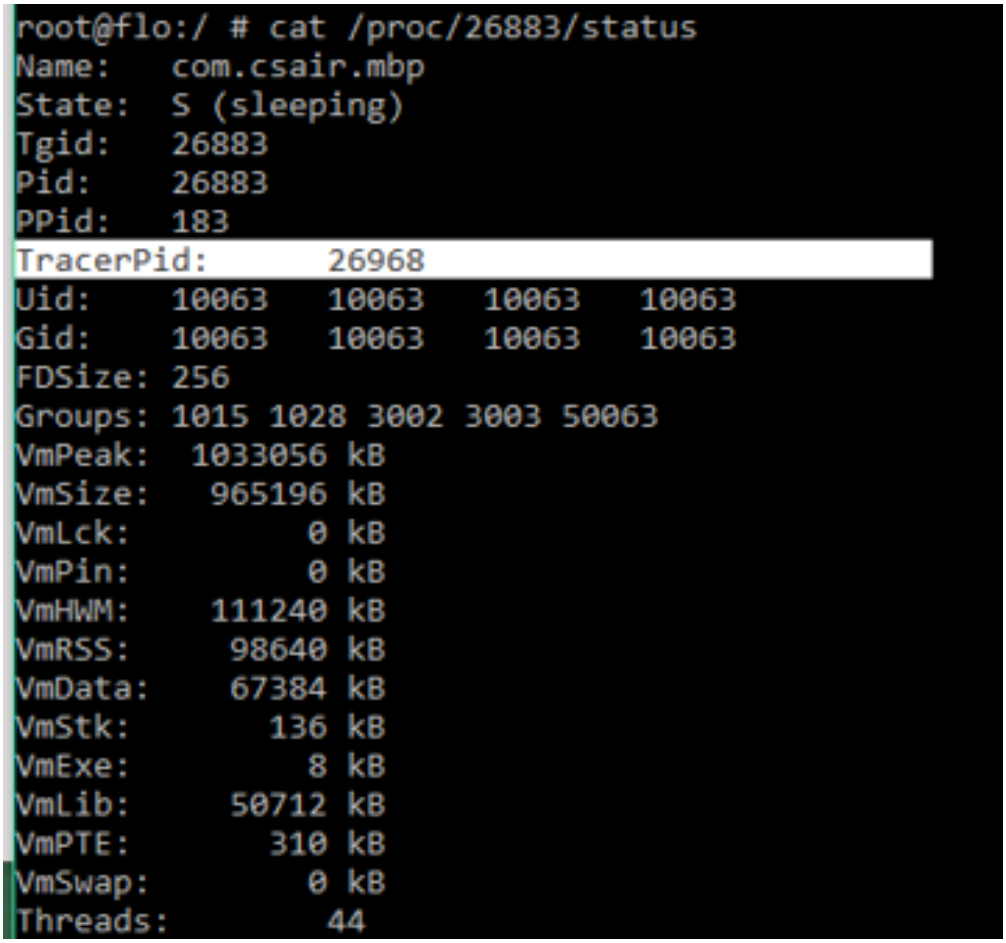


Figure 2.1: Process Status

2.2.2 Anti-emulator

Mobile applications can check the running environment, such that an application can crash or exist if it's running in the emulator or rooted system. Android applications can check several Android APIs to get the system properties or build information (e.g., device brand, hardware, model, device ID, IMEI number, Phone Number etc.) For example, the get phone number API will return null if your application is running inside an emulator.

Anti-emulator techniques employed by malware [10] limits the usage of many dynamic analysis systems. For example, in [11], 98.6% malware samples were successfully analyzed

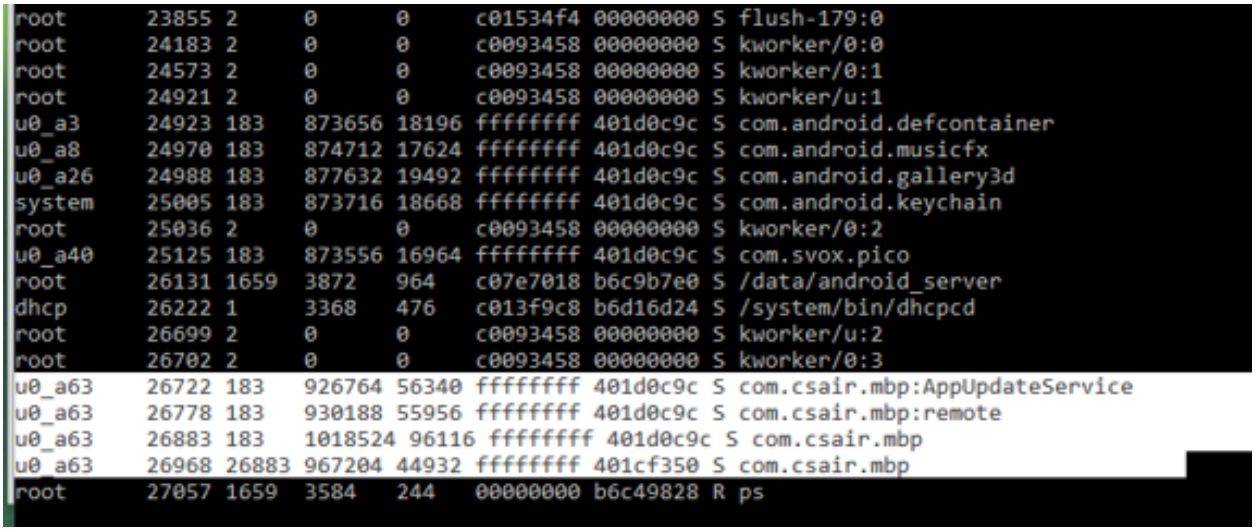


Figure 2.2: Self-attached Anti-debugging

on the real smart-phone, whereas only 76.84% malware samples were successfully inspected using he emulator.

2.2.3 Code Packing

An application is packed means its original executable files (i.e., DEX files) are hidden or transformed to an encrypted or obscured form so that we cannot easily reverse, modify, and repackage.

For Android applications, Java source code is finally compiled to Dalvik byte-code and stored in DEX file. Android allows applications to load codes from external sources at run-time. To leverage this feature, packers usually encrypt original DEX file as an external file, and insert its owner DEX file as a shell or guard. During the execution, packer's decryption stubs, which implemented in native code, will decrypt the original Dalvik byte-code and then load it into memory.

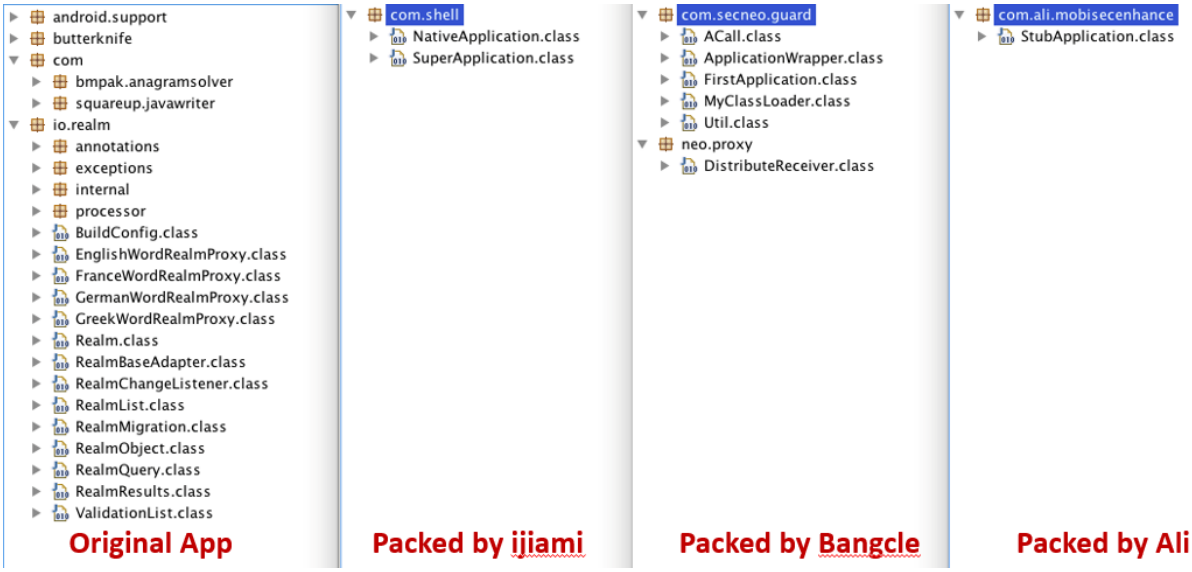


Figure 2.3: Decompiled Code Tree Comparison between Unpacked and Packed Version

Moreover, packer's native components can modify the meta-data of original DEX file through JNI during the execution. This kind of modification doesn't affect the normal execution of the application. However, it significantly affects certain analysis tools.

We investigated several commercial packers that provide on-line packing services (Ali, Bangcle, ijiami, Tencent, etc.), and summarize the common anti-analysis defenses used by them in next chapter. Figure 2.3 shows an example for packed Android applications and unpacked Applications. The left side of the decompiled code tree is for the original application. We can use common DEX decompiler as mentioned previously to extract all the classes. However, after packing, the original classes are hidden, which cannot be extracted by the decompiler, as shown in the rest three decompiled code tree.

2.2.4 Code Obfuscation

Obfuscation aims at creating obfuscated code that is difficult for humans to understand. Obfuscation techniques include modifying names of classes, fields, and methods, reordering

control flow graphs, encrypting constant strings, inserting junk code, etc. Many developers would obfuscated their applications before released. Those techniques mainly include:

- **Identifier Mangling**: renaming class names, method names and variable names as meaningless strings or even non-alphabet Unicode.
- **String Obfuscation**: replacing static-stored strings with dynamic generated ones.
- **Control Flow Flattening**: flatten the control flow graph of a program. Here is an example. Consider this very simple C program in Listing 2.1. The flattening pass will transform this code into Listing 2.2. As one can see, all basic blocks are split and put into an infinite loop and the program flow is controlled by a switch and the variable b. Figure 2.4 shows the original control flow graph for the C program. After the flattening, we get the instruction flow in Figure 2.5. As one can see, the main difference between the example in pure C, the IR (Intermediate Representation) version is completely flattened.

Listing 2.1: Original C Program

```
#include <stdlib.h>
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
    if(a == 0)
      return 1;
    else
      return 10;
  return 0;
}
```

Listing 2.2: C Program After Control Flow Flattening

```
#include <stdlib.h>
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  int b = 0;
  while(1) {
    switch(b) {
      case 0:
        if(a == 0)
          b = 1;
        else
          b = 2;
        break;
      case 1:
        return 1;
      case 2:
        return 10;
      default:
        break;
    }
  }
  return 0;
}
```

- **Instruction Replacing**: using a set of instructions to replace one instruction while keeping the semantic of the replaced instruction. For example, we can replace standard binary operators (like addition, subtraction or boolean operators) by functionally

equivalent, but more complicated sequences of instructions. Table 2.1 shows a list of instruction substitution examples.

Table 2.1: Instruction Replacing

Operation	Original Expression	Instruction Replaced Expression
Addition	a = b + c	a = b - (-c)
Subtraction	a = b - c	a = b + (-c)
AND	$a = b \wedge c$	$a = (b \oplus \neg c) \wedge b$
OR	$a = b \vee c$	$a = (b \wedge c) \vee (b \oplus c)$
XOR	$a = a \oplus b$	$a = (\neg a \wedge b) \vee (a \wedge \neg b)$

- **Junk Code Injection**: injecting useless code to change original control flow. For example, we can modify a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block. The original basic block is also cloned and filled up with junk instructions chosen at random. For example, after the bogus control flow pass, we can change the original C program's Control Flow Graph in Figure **??** to a new flow graph in Figure 2.6.

2.3 Software Instrumentation

As a prerequisite for various performance-analysis and debugging techniques, it is often necessary to insert additional code fragments into the application that is currently under investigation, e.g., to validate parameters given to a function call, read hardware counter values such as the number of cache misses, or query the system clock to calculate the time spent in a certain code region [12]. Software instrumentation is the technique that is widely used in software profiling, performance analysis, optimization, testing, error detection, virtualization, and to write trace information. In programming, instrumentation means the ability of an application to incorporate the following.

- **Code tracing**: receiving informative messages about the execution of an application at run time.

- **Debugging and (structured) exception handling**: tracking down and fixing programming errors in an application under development.

- **Profiling**: a means by which dynamic program behaviors can be measured during a training run with a representative input. This is useful for properties of a program that cannot be analyzed statically with sufficient precision, such as alias analysis.

- **Performance counters**: components that allow the tracking of the performance of the application.

- **Computer data logging**: components that allow the logging and tracking of major events in the execution of the application.

Instrumentation, which involves adding extra code to an application for monitoring some program behavior, can be performed either statically (i.e., at compile time) or dynamically (i.e., at runtime). Static instrumentation techniques range from simple manual techniques to compiler/assemblerbased instrumentation and linktime or postlink executable editing. Dynamic instrumentation techniques are often more complex to implement than the static ones, but they can track dynamically linked libraries and indirect branches that are difficult to handle through static instrumentation [13].

Developers implement instrumentation in the form of code instructions that monitor specific components in a system (for example, instructions may output logging information to appear on the screen). When an application contains instrumentation code, it can be managed by using a management tool. Instrumentation is necessary to review the performance of the application. Instrumentation approaches can be of two types: source code instrumentation and binary instrumentation.

2.3.1 Source Code Instrumentation

A well-accepted technique of instrumenting an application is the so-called source code instrumentation method, which is the subject matter of this paper. With this approach, additional code fragments such as function calls are directly inserted into the applications source code at appropriate places before compilation [12]. Our Android packer analysis system makes extensive use of source code instrumentation to transparently monitor the execution behavior and extract class and method information on both byte-code level and native level. We insert extract code into functions that perform classes loading, class resolve and method resolve, etc. to capture information that can be used for DEX file reassemble.

2.3.2 Binary Instrumentation

Section 2.1 explained that dynamic analysis requires programs to be instrumented with analysis code. There are two main instrumentation techniques used for dynamic binary analysis, which are distinguished by when they occur [14].

- **Static binary instrumentation** occurs before the program is run, in a phase that rewrites object code or executable code.
- **Dynamic binary instrumentation**: occurs at run-time. The analysis code can be injected by a program grafted onto the client process, or by an external process. If the client uses dynamically-linked code the analysis code must be added after the dynamic linker has done its job.

Dynamic binary instrumentation has two distinct advantages. First, it usually does not require the client program to be prepared in any way, which makes it very convenient for users. Second, it naturally covers all client code; instrumenting all code statically can be difficult if code and data are mixed or different modules are used, and is impossible if the client uses dynamically generated code. This ability to instrument all code is crucial for correct and complete handling of libraries. These advantages of dynamic binary instrumentation make it

the best technique for many dynamic analysis tools. However, dynamic binary instrumentation has two main disadvantages. First, the cost of instrumentation is incurred at run-time. Second, it can be difficult to implement rewriting executable code at run-time is not easy. Nonetheless, in recent years these problems have been largely overcome by the advent of several generic dynamic binary instrumentation frameworks, which are carefully optimized to minimize run-time overheads, and with which new dynamic binary analysis tools can be built with relative ease. Our function similarity mapping framework, MobileFindr, focuses on dynamic binary instrumentation, and does not consider static binary instrumentation any further. MobileFindr instruments system level libraries and frameworks to capture various dynamic behavior features during the execution of a function along a runtime trace. Then we calculate the similarity distance based on such features to perform function similarity mapping.

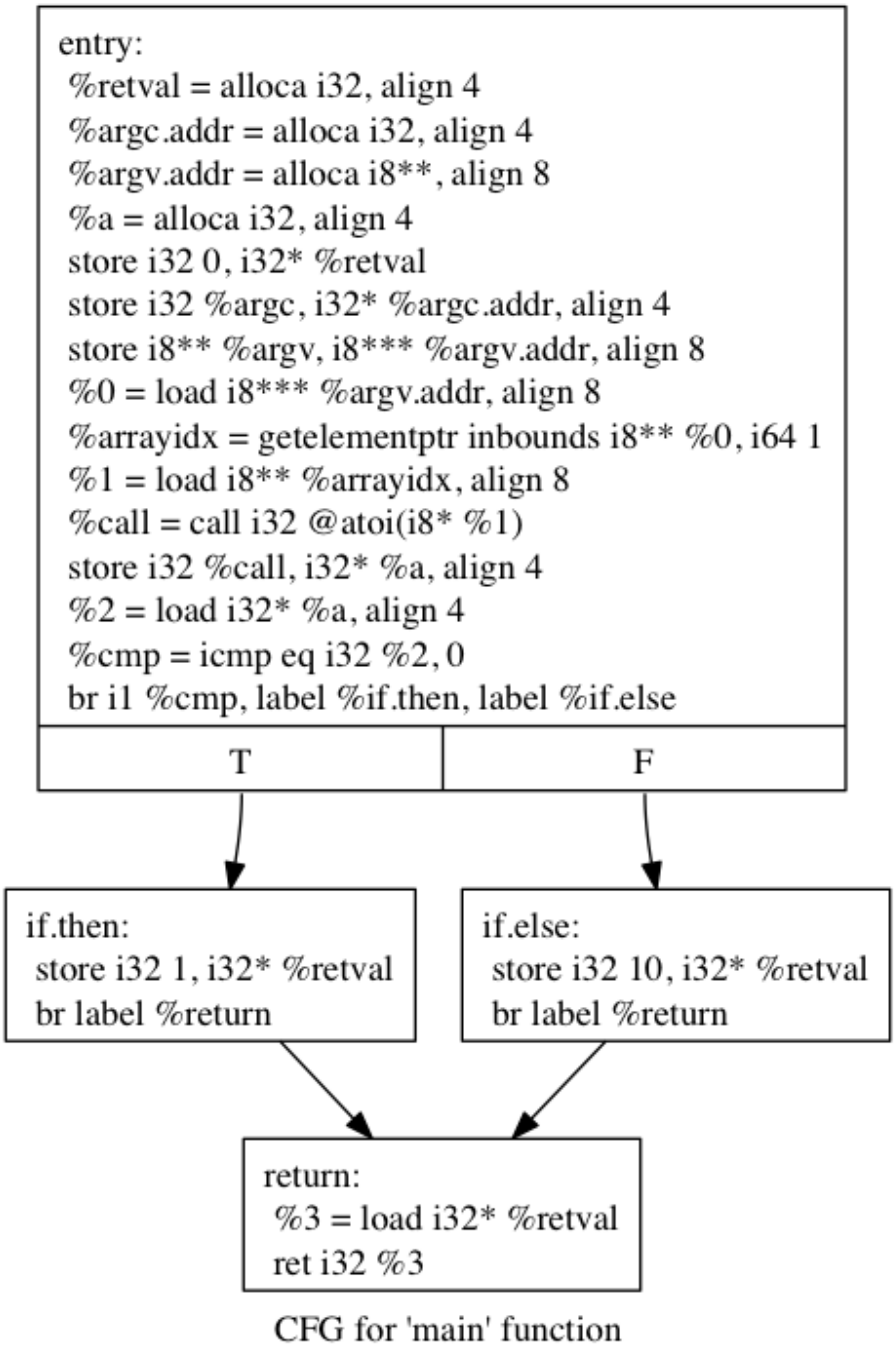


Figure 2.4: Original Control Flow Graph

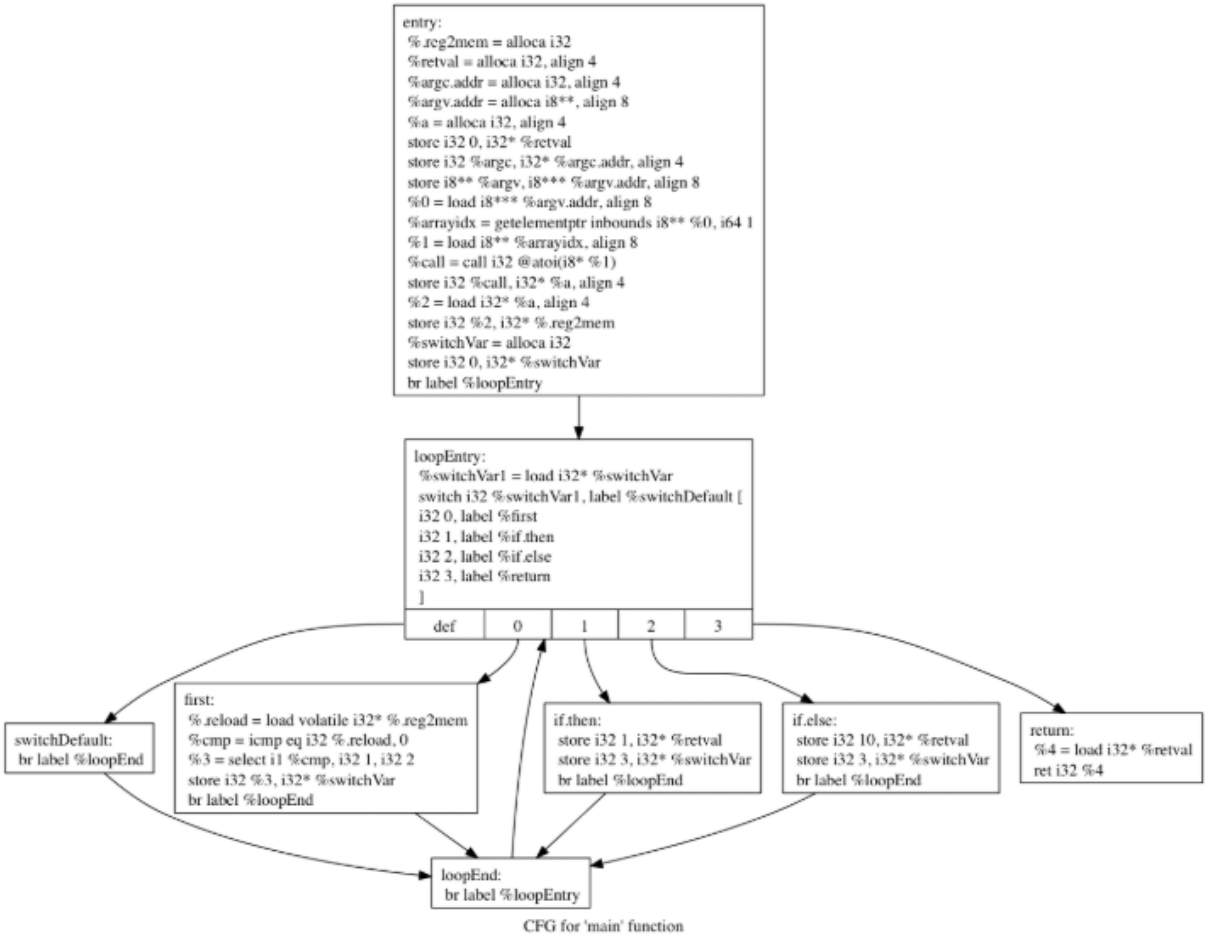


Figure 2.5: Control Flow Graph After Flattening

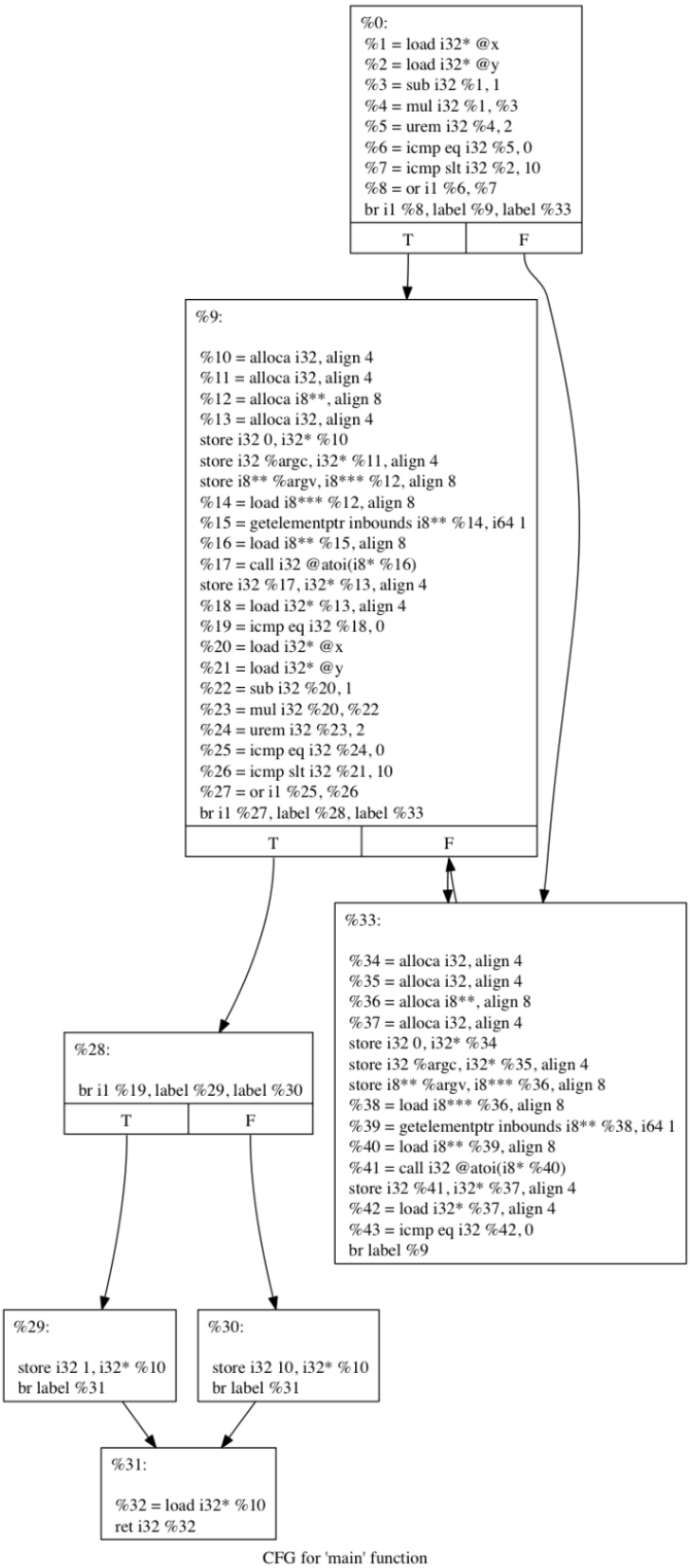


Figure 2.6: Bogus Control Flow

Chapter 3

System Instrumentation for Analyzing Packed Android Applications

3.1 Introduction

Being the most popular mobile operating system, Android dominated smart-phone market with a share of 82.8% until 2015 [15]. The rapid growth of Android application economy brings a great profits for developers [16], meanwhile, it causes a series of malicious tampering, code injection, and plagiarism issues [17, 18, 19]. Legitimate application developers adopt various code protection techniques to guarantee their labor and profits. Packing is one of the effective and efficient code protection techniques, and is getting an increasingly use nowadays [20].

Although packing techniques are initially designed to protect applications from being reversed, modified, and repackage, malware writers are making use of these benefits to hide their malicious code in order to evade malware detection. A huge growing percentage of packed Android malware has been reported by the AVL anti-virus team [21]. Since packers usually adopt complex anti-analysis defenses, recent anti-virus could not perform effective analysis task on packed code, and thus are not able to detect those packed malware automatically. Therefore, a number of unpacking approaches have been proposed recently [22, 23, 24].

All of the unpacking approaches are focusing on byte-code level analysis that recover original Dalvik byte-code from memory. For example, [24] mainly focus on dumping the loaded DEX file in memory directly to recover the original DEX file. DexHunter [23] exploits the class loading process of Android's virtual machine to recover the DEX files from packed applications. AppSpear [22] also focus on reassemble the DEX file by collecting the Dalvik Data Structure in memory. However, packers are evolving frequently, most unpacking approaches

only work for a limited time or for a particular type of packers. Advanced packers employ multi-layer packing techniques, which make the current unpacking techniques ineffective.

In this work, we designed an automatic analysis platform that provides a comprehensive view of packed Android applications behavior by conducing multi-level monitoring and information flow tracking.

- **Byte-code level analysis**: instruments both Android Runtime (ART) and Dalvik Virtual Machine (DVM) to extract the hidden class information during the applicationss execution. it leverages both class loading and method resolving process at runtime to capture the Dalvik byte-code. this analysis part is capable of identifying packed applications automatically, and then reassemble the original DEX files that was hiding by the packer.

- **Native code level analysis**: monitors the execution of native components in packed Android applications. This monitoring analysis can be used to reveal the behavior of a packer. These frameworks include system call monitoring, Native-to-Java communication monitoring through JNI trace, library calls monitoring (libc trace), and IPC transaction monitoring through binder.

We have evaluated our system with a set of open source Android examples packed by different major on-line packers. Our experimental results show that our system can successfully detect and extract the hidden Dalvik byte-code, and reassembly to their original DEX files. Our evaluation with real-world mobile applications also demonstrated the effectiveness of our native code monitoring process.

Correspondingly, our contributions in this work are:

- We proposed a novel unpacking method that can detect packed application from unpacked one automatically, and extracting the hidden Dalvik byte-code from recent multi-layer packing technique.

- We proposed a JNI instrumentation approach to trace the Java-to-Native and Native-to-Java invocation. This monitoring process can be used to analyze the behavior of packers since the packer's decryption process is mainly done by native code.

- we have implemented a framework for this work and source code is publicly available at GitHub: https://github.com/tigerlyb/android_packing_analysis.

- We have demonstrated the viability of our approach for real-world mobile applications.

3.2 Background

This section introduces the background of how Android Applications are built and run. We also introduce Android Dalvik Virtual Machine (DVM) and the new Android Runtime (ART) as well as the Java Native Interface (JNI). In addition, we present the basic idea of Android packing as well as the state of the art techniques employed by packer. Then we demonstrate motivating examples and describe the unpacking challenges that can affect the state of the art Android application analysis methods.

3.2.1 How Android Applications are Built and Run

Figure 3.1 shows the process of building an Android application.

- The Android Asset Packaging Tool (aapt) takes your application resource files, such as the AndroidManifest.xml file and the XML files for your Activities, and compiles them. An R.java is also produced so you can reference your resources from your Java code.

- The aidl tool converts any .aidl interfaces that you have into Java interfaces.

- All of your Java code, including the R.java and .aidl files, are compiled by the Java compiler and .class files are output.

- The dex tool converts the .class files to Dalvik byte-code. Any 3rd party libraries and .class files that you have included in your module build are also converted into .dex files so that they can be packaged into the final .apk file.

- All non-compiled resources (such as images), compiled resources, and the .dex files are sent to the apkbuilder tool to be packaged into an .apk file.

- Once the .apk is built, it must be signed with either a debug or release key before it can be installed to a device.

- Finally, if the application is being signed in release mode, you must align the .apk with the zipalign tool. Aligning the final .apk decreases memory usage when the application is -running on a device.

3.2.2 Dalvik Virtual Machine

The Dalvik Virtual Machine (DVM) is an Android virtual machine optimized for mobile devices. Android version 4.0 and lower use the Dalvik virtual machine (DVM) with just-in-time compilation to run Dalvik byte-code, which is usually translated from Java byte-code. With the Dalvik JIT compiler, each time when the application is run, it dynamically translates a part of the Dalvik byte-code into machine code. As the execution progresses, more byte-code is compiled and cached. Since JIT compiles only a part of the code, it has a smaller memory footprint and uses less physical space on the device.

Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.

Basically, the Dalvik virtual machine performs transformation of the applications Dalvik byte-code into native instructions, so there would be platform-specific Dalvik virtual machines for each hardware platform running Android, be it from Intel, ARM, or TI. A developer's compiler creates Dalvik byte-code, and a Dalvik virtual machine deciphers that byte-code.

3.2.3 Android Runtime

Android Runtime (ART) is a new application runtime environment being introduced experimentally in the Android version 4.4 release, and becomes the default runtime from version 5.0. Unlike Dalvik, ART introduces the use of ahead-of-time (AOT) compilation by compiling entire applications into native machine code upon their installation. By eliminating Dalvik's interpretation and trace-based just-in-time (JIT) compilation, ART improves the overall execution efficiency and reduces power consumption, which results in improved battery autonomy on mobile devices. At the same time, ART brings faster execution of applications, improved memory allocation and garbage collection (GC) mechanisms, new applications debugging features, and more accurate high-level profiling of applications.

ART comprises a compiler (the **dex2oat** tool) and a runtime (libart.so) that is loaded for starting the Zygote. Zygote is a special process in Android which handles the forking of each new application process. The **dex2oat** tool takes an APK file and generates one or more compilation artifact files that the runtime loads. The number of files, their extensions, and names are subject to change across releases, but as of the Android O release, the files being generated are:

- **.vdex**: contains the uncompressed DEX code of the APK, with some additional metadata to speed up verification.
- **.odex**: contains AOT compiled code for methods in the APK.

- **.art (optional)**: contains ART internal representations of some strings and classes listed in the APK, used to speed application startup.

The **.odex** file has an *oatdata* section, which contains the information of each class that has been compiled into native code. The native code resides in a special section with the offset indicated by the *oatexec* symbol. Hence, we can find the information of a Java class in the *oatdata* section and its compiled native code through the *oatexec* symbol.

When an application is launched, the ART runtime parses the **.odex** file and loads the file into memory. For each Java class object, the ART runtime has a corresponding instance of the C++ class *Object* to represent it. The first member of this instance points to an instance of the C++ class *Class*, which contains the detailed information of the Java class, including the fields, methods, etc. Each Java method is represented by an instance of the C++ class *ArtMethod*, which contains the methods address, access permissions, the class to which this method belongs, etc. The C++ class *ArtField* is used to represent a class field, including the class to which this field belongs, the index of this field in its class, access rights, etc. We can leverage the C++ *Object*, *Class*, *ArtMethod* and *ArtField* to find the detailed information of the Java class, methods and fields of the Java class [25].

3.2.4 Java Native Interface

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native components and libraries written in other languages such as C, C++ and assembly.

JNI enables programmers to write native methods to handle situations when an application cannot be written entirely in the Java programming language, e.g. when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application (written in another programming language) to be accessible to Java applications. Many of the standard library classes depend on JNI to provide functionality to the developer and the user, e.g. file I/O and sound capabilities.

Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner.

The JNI framework lets a native method use Java objects in the same way that Java code uses these objects. A native method can create Java objects and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code.

JNI defines two key data structures, "JavaVM" and "JNIEnv". Both of these are essentially pointers to pointers to function tables. (In the C++ version, they're classes with a pointer to a function table and a member function for each JNI function that indirects through the table.) The JavaVM provides the "invocation interface" functions, which allow you to create and destroy a JavaVM. In theory you can have multiple JavaVMs per process, but Android only allows one.

The JNIEnv provides most of the JNI functions. The native functions receive a JNIEnv as the first argument. JNIEnv is used for thread-local storage. For this reason, we cannot share a JNIEnv between threads. If a piece of code has no other way to get its JNIEnv, you should share the JavaVM, and use GetEnv to discover the thread's JNIEnv.

The C declarations of JNIEnv and JavaVM are different from the C++ declarations. The "jni.h" include file provides different typedefs depending on whether it's included into C or C++. For this reason it's a bad idea to include JNIEnv arguments in header files included by both languages.

JNI general work-flow is as follows [26]: Java initiates calls so that the local function's side code (such as a function written in C/C++) runs. This time the object is passed over from the Java side, and run at a local function completion. After finishing running a local function, the value of the result is returned to the Java code. Here JNI is an adapter, completing mapping between the variables and functions (Java methods) between the Java language and native compiled languages (such as C/C++). We know that Java and C/C++

are very different in function prototype definitions and variable types. In order to make the two match, JNI provides a jni.h file to complete the mapping between the two. This process is shown in Figure 3.2.

3.2.5 Android Packing Services

An application is packed means its original executable files (i.e., DEX files) are hidden or transformed to an encrypted or obscured form so that we cannot easily reverse, modify, and repackage. Figure 3.3 shows the decompiled code from an unpacked Android application. Comparing to the unpacked one, the method content of packed version is empty.

Figure 3.5 shows the DEX code content dumped directly from memory. The left side is for unpacked version. The right side is for packed version. All DEX code content has been set to 0.

Code Obfuscation

Obfuscation aims at creating obfuscated code that is difficult for humans to understand. Obfuscation techniques include modifying names of classes, fields, and methods, reordering control flow graphs, encrypting constant strings, inserting junk code, etc.

Packers usually implement obfuscation for major functions in native code, and then invoke native code method through Java native interface (JNI). In addition, Android packers hide method invoking using Java reflection mechanism and use *goto* to make control flow hard to understand [22]. Figure 3.6 shows the code obfuscation in native side. All method names are set to meaningless and are implemented in native code.

Dynamic Code Modification

Android applications are mostly written in Java and then turned into Dalvik byte-code. Note that it is not easy for applications in Dalvik byte-code to arbitrarily modify itself in Dalvik Virtual Machine (DVM) in a dynamic manner. Instead, they can invoke native code through

JNI to modify byte-code in DVM because the native code is running in the same context as the applications DVM so that the native code can access and manipulate the memory storing the byte-codes.

As an example, malware can employ native code to generate malicious byte-codes dynamically and then execute them in DVM. Before executing the DEX file in the new Android runtime (i.e., ART), ART will compile the DEX file into oat file in the ELF format. The native codes in .so files can not only change instructions in DEX and OAT files, but also modify key data structures in the memory, such as DexHeader, ClassDef, ArtMethod, etc., in order to assure that the contents are correct only when they are used, and the contents will be wiped out after they have been used [23].

Dynamic Loading

Android allows applications to load codes from external sources (in DEX or JAR format) at runtime. Leveraging this feature, packers usually encrypt the original DEX file, decrypt and load it before running the application.

3.3 System Design and Implementation

While inspecting the implementation of Android VM environments, we found that we can observe obscured application behaviors through Android VM instrumentation. These obscured application behaviors, such as hidden code, are commonly used by malicious applications. Detecting these behaviors at the OS level is often difficult due to anti-debugging, anti-emulator, etc. which measure intentionally adopted by attackers. Through our multi-level system instrumentation, we can compare the class declarations with the class implementation used by the application at run-time. To recover the hidden code, we automatic trigger the DEX loading process of the VM and extract all class information from memory, and reassembled based on the DEX format by our system.

3.3.1 Byte-code Level Analysis

Most of detection approaches for packed applications are based on manual analysis such as identifying unique file names and native .so libraries inserted by packer. Manual analysis is slow and tedious, which only works for known packers. We propose to design a detection method that automatically identify packed applications. A prototype of the hidden code detection and extraction through VM instrumentation is presented in this section.

One of the major challenges of this work is to evade the complex anti-defenses adopted by packers. To thwart this challenge, we perform both static and dynamic analysis on each application, and compare the difference between the two analysis to determine whether this application is packed. Figure 3.7 shows the overview of detection framework.

Static Analysis

In the static analysis phase, We first extract package name and launchable-activity name from the application. The package name is used as a filter in dynamic analysis and will be sent to the device. The launchable-activity name is used to start the application automatically. Then we use the de-compiler tool *Baksmali* to parse the application and convert its DEX files to a series of smali files, which contain human readable assembly language representing Dalvik byte-code. Finally we extract the class name of each class from the smali files.

Dynamic Analysis

In the dynamic analysis phase, we instrument Dalvik Virtual Machine (DVM) to monitor the execution of this application. This monitoring is a compilation time code injection instrumentation. We didn't modify any of the APIs, only insert our own code to capture the class loading information. Thus, it's very difficult to be aware of by packers' anti-analysis measures. We also deployed the framework on a standard Android device, which can evade typical emulator detection of packers. This guarantees a very trustworthy analyzing environment.

Since each class should be loaded into memory before it can be used, we explore the class loading process to collect class information at runtime. Figure 3.8 shows the general class loading process at runtime. Android VM first will unzip the apk file and look for DEX files, then it opens and parses this DEX file. After that Android VM will initialize all the classes defined in this DEX file and load all classes into memory. In order to automatically capture the class at runtime, we instrument the class loading functions defined in both DVM and ART.

We select *DexFile_defineClassNative* in DVM and *ClassLinker::DefineClass* in ART as the key functions for injecting instrumentation code. DEX file is parsed into a data structure called *DvmDex*, and then initialized to *DexFile* in memory. Since all class loadings will call this function, including applications running background, we first implement a filter based on the package name captured in static analysis and search the *p_id* of the application to be analyzed. After locating the correct *DexFile* object with the *p_id*, we obtain the class index object *DexClassDef* by passing *DexFile* as an parameter of the *dexGetClassDef* function, and then invoke the method *dexGetClassDescriptor* to extract the class name of each class.

Comparison

Since the class name is unique, we only extract full class name as a string for each class in the static and dynamic analysis. We sort the class name strings in order and then compare their difference from the two analysis. If we find class names that only exist in dynamic analysis, this means the application contains packed code. Otherwise, the application is unpacked.

DEX recovery

As mention previously, advanced packer has multi-layer packing. First, the original DEX file will be hidden and will be release at runtime. However, the content of the method is empty even if the DEX file has been loaded in the memory. The content will only be loaded when the method has been invoked. After the instrumentation of JNI (Java Native Interface),

we detected frequent native-to-Java calls with *dvmResolveClass()* and *dvmResolveMethod()* when execution the packed applications but no such calls or a little bit invocation in unpack version. *dvmResolveMethod()* is used for reload the method if this method is not initialized properly. Packer invoke this Android API to reloaded method and then erase the content of that method after use in order to prevent the direct memory dump based unpacking method.Therefore, we instrument both *dvmResolveClass()* and *dvmResolveMethod()* to capture the class information and dump method content. The first row of Table 3.1 shows all instrumented functions in DVM and ART.

3.3.2 Native Code Level Analysis

Native code in Android applications is deployed in the application as ELF files, either executable files or shared libraries. Android Packers usually exploit native code to perform dynamic code loading and dynamic code modification. During the dynamic analysis phase we monitor the execution of native components as well as the communications between the Java code and the native code via JNI instrumentation. This monitoring analysis can be used to reveal the behavior of a packer. As mentioned in previous section, our JNI instrumentation framework capture frequent native-to-Java calls with *dvmResolveClass()* and *dvmResolveMethod()* when execution the packed applications but no such calls or a little bit invocation in unpack version. Table 3.1 shows all instrumented functions that used for monitoring the behavior of packer. From the second row of Table 3.1, native code loading means that the code that enable hidden byte-code release could be implemented in native code and loaded into memory. Note that Android packers usually exploit these APIs to directly load the decrypted byte-code from memory. JNI invocation refers to all the function calls from Java methods to native methods. This includes the JNI calls in the application and the Android framework. JNI reflection, on the other hand, refers to calling Java methods from native. For instance, packer's decrypting code stub implemented in native code could invoke framework APIs using JNI reflection.

Table 3.1: Complete Instrumented Functions in DVM and ART

Behavior	Functions in DVM	Functions in ART
Java class loading	DvmDefineClassNative() DvmResolveClass() DvmResolveMethod()	DexFile::DexFile() DexFile::OpenMemory() ClassLinker::DefineClass()
Native code loading	dvmLoadNativeCode()	JavaVMExt::LoadNativeLibrary()
JNI invocation	RegisterNatives() dvmCallJNIMethod()	artFindNativeMethod() ArtMethod::invoke()
JNI reflection	dvmCallMethodA() dvmCallMethodV	InvokeWithVarArgs() InvokeWithJValues() InvokeVirtualOrInterfaceWithJValues() InvokeVirtualOrInterfaceWithVarArgs()

3.4 Evaluation

In this section, we evaluate our system to answer the following questions.

- **Q1**: Can our framework detect and recover the DEX file from packed Android applications?
- **Q2**: Can our framework analyze the packed Android applications in real world?

3.4.1 Experiment Setup

As shown in Figure 3.7, detection module takes a set of Android applications as input, and output a set of packed applications. The static analysis is performed in a desktop computer running Ubuntu Desktop 14.04. We use Baksmali 2.1.0 as a basis for the static analysis. The instrumented DVM is deployed in a Nexus 7 tablet running Android 4.4.3. In the dynamic analysis, applications will be automatically installed and launched by triggering the launchable-active defined in the manifest file. To save the storage space, applications will be automatically deleted after class names have been extracted.

3.4.2 Datasets

Dataset 1

First we collect 20 open source Android applications with different functionalities from F-Droid [27] and uploaded them to 5 major online packers: Ali, Bangcle, Tencent, ijiami, and qihoo360. Since the source code is available, we use this dataset as ground truth dataset. Table 3.2 shows that not all applications can be packed by those packers and some of the packed applications cannot be run in our devices. Therefore, we built 81 packed application in total.

Table 3.2: Building Packed Android Applications

Packers	Number of Apps	Number of Packed Apps	Number of Packed Apps Can Run
Bangcle	20	20	20
ijiami	20	19	19
Ali	20	18	16
Tencent	20	16	10
qihoo360	20	17	16

Dataset 2

Second, we collect 822 Android applications from anzhi (http://www.anzhi.com/), one of the most popular Android market in China. We use a web crawler to automatically download top-ranked apk files from the market.

3.4.3 Results

The first evaluation method is to answer **Q1**. Dataset 1 is used for this evaluation. We apply our framework to all packed applications that can run in our device. After the DEX recovery, we use baksmali tools to decompiled the DEX file and compare the byte-code with the original unpacked version. We have successfully recovered all 81 packed applications.

The second round of evaluation method worked on the Dataset 2, which is to answer **Q2**. We define an application is packed if we can capture more class in the dynamic analysis

than static analysis. Table 3.3 shows the detection results. There are 38 applications failed to run automatically since there is no launchable activities defined in the manifest file. There are 646 applications are unpacked since we extracted exactly same number of classes from dynamic and static analysis. All class names are same as well. After manual verification, we have 32 false positives because of the disassembling error by baksmali tools. E.g., we extract 1605 classes in the static analysis and extract 1638 classes in the dynamic analysis. There are 35 classes that cannot be decompiled by baksmali, but the main body of this application can be decompiled. [28] shows a complete analysis result of class extraction for Dataset 2.

Table 3.3: Detecting Packed Android Applications in Real World

	Number of Apps	Comments
Total Apps	822	
Unpacked	646	static = dynamic
Packed	138	static < dynamic
Failure	38	Auto-run failed

3.5 Discussion and Future Work

With different types of packers, and the vast range of packed applications within each type, it's important that every packed application can be easily distinguished and unambiguously classified. Therefore, we can design an automatic classification framework to identify and classify packed applications as future work. Since we have identified which Android applications are packed, we can categorize them into groups that reflect similar types of behaviors. We propose a two-layer classification, coarse-grained classification and fine-grained classification as following:

- **Coarse-grained Classification**: In coarse-grained classification, we can look at which part of code is packed. Based on our observation, many applications contain third-party libraries and only the libraries are packed. We consider this type of packed applications is partially packed or framework packed. The application which its entire

code is packed will be considered fully packed. For the fully packed applications, we can extract much more classes in the dynamic analysis than static analysis. Such that we extract 4806 classes in dynamic analysis but only 6 classes in static analysis. We evaluated 200 fully packed applications, more than 99% of them has the ratio (number of classes in static / number of classes in dynamic) less than 20%. Thus, we can set a threshold as 20% to classify framework packed and fully packed applications. If the packed application is framework packed and the framework libraries are from well-known legitimate publishers, we may ignore the analysis for that application.

- **Fine-grained Classification**: As mentioned previously, packers usually encrypt original DEX file to external data format, insert decryption stubs and customized loader into the application. The original byte-code of the application will be released during the execution by the decryption stub and loaded into memory by packer's customized loader. Different packers may implement different code releasing and loading strategies. For instance, some packers reload the DEX data into in-consecutive memory regions and modify relevant pointers that point to the data to prevent direct memory dump based unpacking; a type of packers deploy a two-layer decryption stub. It first releases a decrypted DEX, which doesn't contain the original byte-code. However, it contains a second decryption stub responsible for decrypting original byte-code of a method once it's invoked [22].

All behaviors performed above can be eventually represented as a sequence invocation of native activities, system calls, JNI calls, etc. Packers implement similar code release or loading strategy may result similar behavioral pattern. Such as a similar pattern of system calls, or a similar pattern of JNI calls, which make these packers belong to one type of packer. Native code in Android applications is deployed as ELF files, either executable files or shared libraries (.so files). Java code can invoke native code in the following ways [29].

Exec methods. Executable file can be called from Java by *Runtime.exec* and *process-Builder.start*. These methods are refer to Exec methods.

Native methods. Methods are declared in Java code but implemented in native shared libraries. Java Native Interface (JNI) defines a way for Java to interact with native methods.
Native activity. Native code is invoked in native activities using activities' callback functions, (e.g., *onCreate* and *onResume*), if defined in a native library.

To address the diversity of packers' behavior and detect the typical behavioral patterns for specific type of packer, we propose to design a native code level instrumentation framework that records all events and operations executed from within native code, such as invoked system calls, native-to-Java communications and Binder transactions as following:

- To monitor system calls, we proposed to implement a Linux kernel module to capture the invoked system calls. Other tools such as *strace* cannot perform effective analysis because of the packer's anti-debugging.

- To monitor native-to-Java communications including calls to Exec methods, calls to Native methods, native activity callbacks, we proposed to instrument *libjavacore*, *libdvm*, *libandroid_runtime* respectively.

- To monitor Binder transactions, we propose to instrument *libbinder* to capture the class of the remote functions being called and the number that identifies the function.

To classify the packed applications, we can use similar idea we proposed in [30]. First, for each application instance i, we can convert the behavior patterns into sequential strings according to the execution time, say sequence for system call monitoring $sc_{\{i\}}$, sequence for Native-to-Java monitoring $nj_{\{i\}}$, and sequence for binder transaction monitoring $bt_{\{i\}}$. Such as $S_{sc_{\{i\}}}$, $S_{nj_{\{i\}}}$, and $S_{bt_{\{i\}}}$. Take system call monitoring for example, if the system call sequence of ith instance $sc_{\{i\}}$ is read→read→wirte, then $S_{sc_{\{i\}}}$ can be written as "RRW". The normalized Levenshtein distance can be then used to compute the distance between behavior patterns of application instance i and k. We define the normalized Levenshtein distance as following. If we have two strings S_1 and S_2, the normalized Levenshtein distance $D(S_1, S_2)$ equals to the minimal operations taken to transform S_1 to S_2, divided

by $max(length(S_1), length(S_2))$. For example, if $S_{sc_{\{i\}}}$ ="RRW" and $S_{sc_{\{k\}}}$ ="RRWW", one operation will be taken to transform $S_{sc_{\{i\}}}$ to $S_{sc_{\{k\}}}$, i.e. adding an extra "W". Then we have the normalized Levenshtein distance between $S_{sc_{\{i\}}}$ and $S_{sc_{\{k\}}}$, $D(S_{sc_{\{i\}}}, S_{sc_{\{k\}}}) = \frac{1}{max(3,4)} = \frac{1}{4}$. Finally, the overall distance between application instance i and k will be $D_{total_{\{i,k\}}} = w_{sc} \cdot D(S_{sc_{\{i\}}}, S_{sc_{\{k\}}}) + w_{nj} \cdot D(S_{nj_{\{i\}}}, S_{nj_{\{k\}}}) + w_{bt} \cdot D(S_{bt_{\{i\}}}, S_{bt_{\{k\}}})$, where w_{sc}, w_{nj}, and w_{bt} is the corresponding weights applied to distances of behavior patterns of system call, Native-to-Java and binder transaction. Given n application instances, a $n \times n$ distance matrix between each application instance is built in this way and single-linkage hierarchical clustering algorithm can be applied to split application instances into clusters (sets of packed applications).

For the coarse-grained classification, we can maintain a white-list database for existing known packed frameworks as a filter. The unknown framework packed applications and fully packed applications will be further classified in the following fine-grained classification. All three monitor components will be deployed in another Android device. Each monitor component will log the corresponding behavior pattern of the packed application and feed the pattern to the classifier. Packed applications will be classified into groups reflecting similar type of behavior eventually.

3.6 Conclusion

In this work, we investigated a series of Android packing services appeared recently, studied the packing techniques adopted by packers and the difference between packed applications and unpacked applications. Our study showed that the complexity of packing techniques and packers' evolvement makes many of the packed applications hard to be detected and analyzed efficiently with existing tools.

Based on our findings, we implemented a detection and DEX recovery module for packed Android applications by combining static and dynamic analysis to evade the anti-defenses of packer, and proposed class load and method resolving instrumentation to capture the

Dalvik byte-code at runtime. In addition, we perform native code level instrumentation to extract information for analyzing native code behavior. Currently more and more Android applications are written in native code in order to speed up execution. Packers are more likely to employ native code packing techniques to protect mobile binary being reversed.

We proposed coarse-grained and fine-grained Android packer classifications as future work. This classification framework aims to extract the discriminative behavior pattern and categorizes packed Android applications into groups.

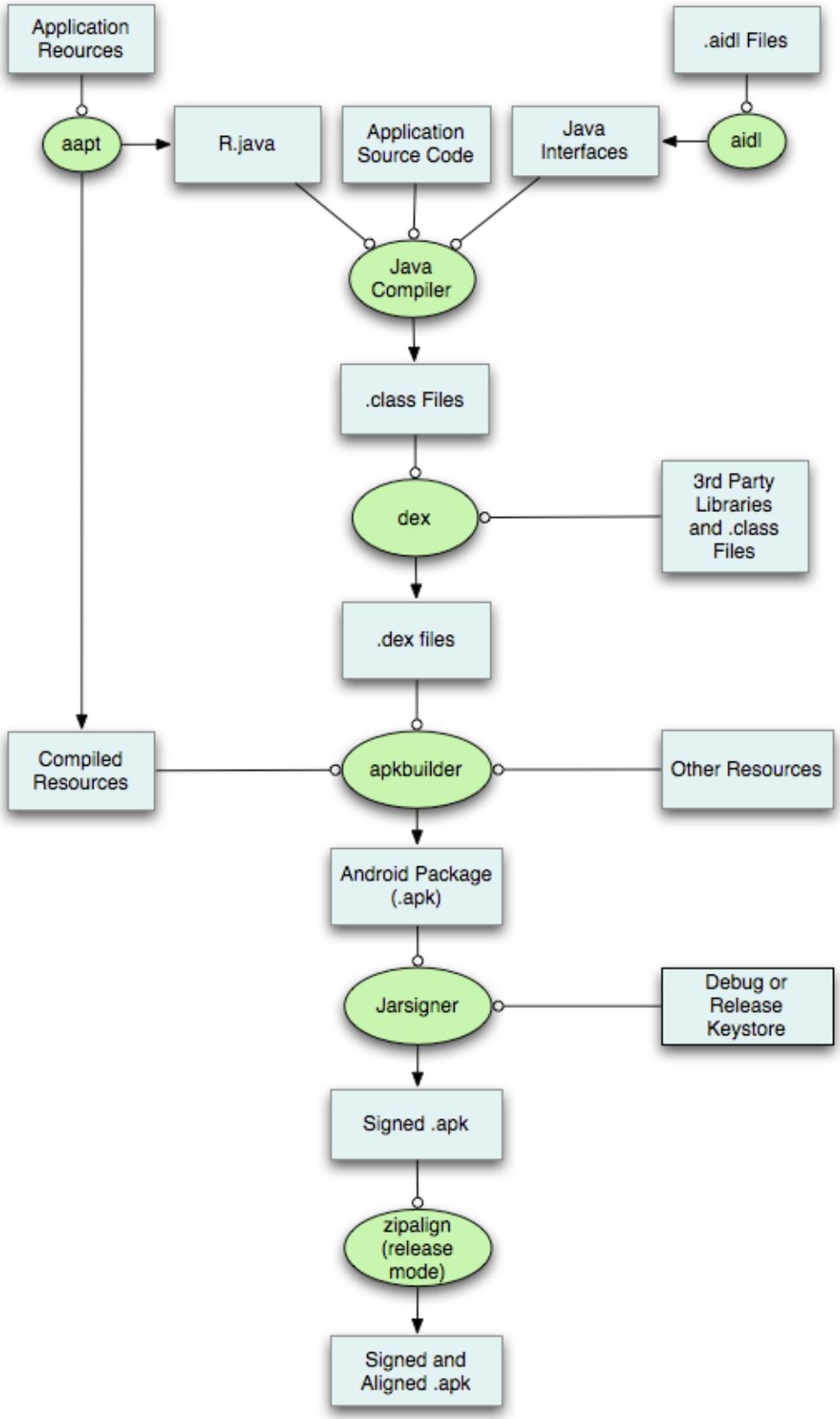


Figure 3.1: How Android Applications are Built and Run

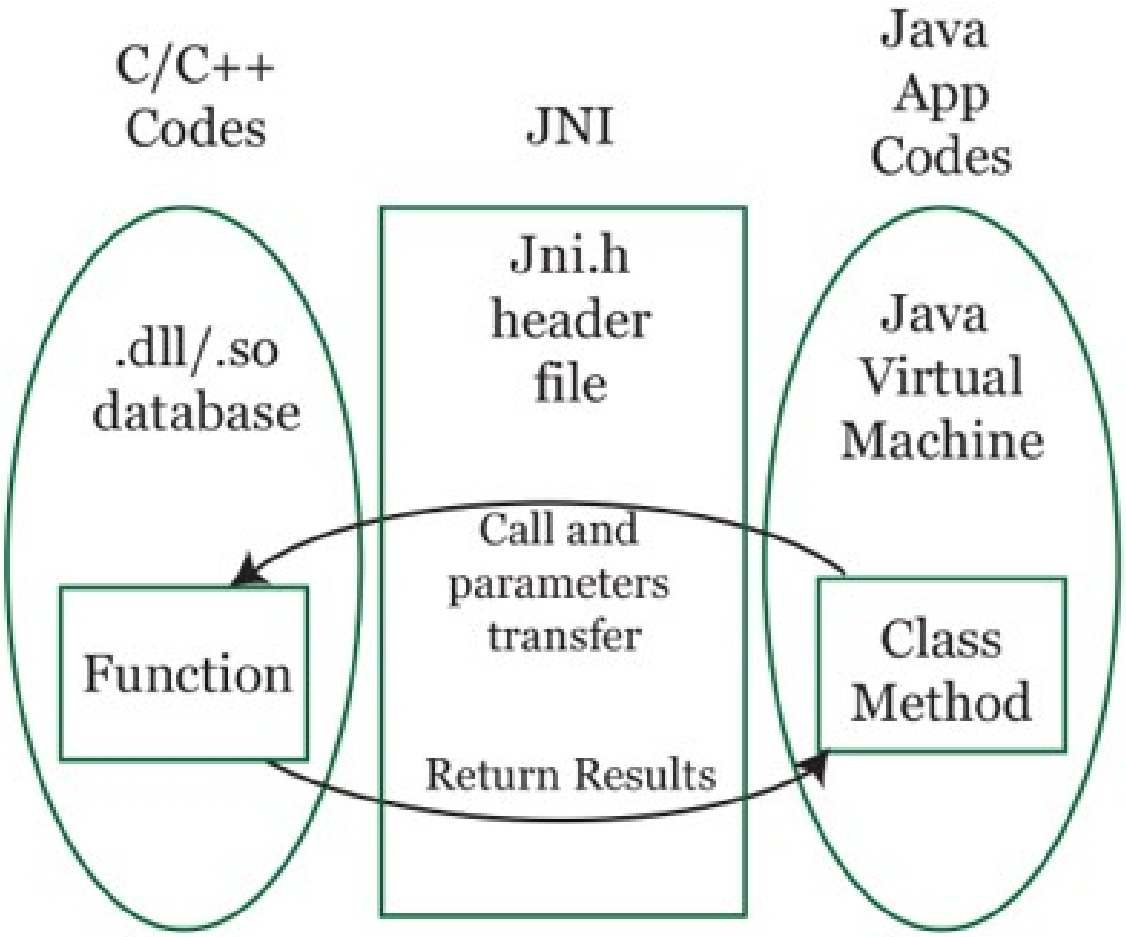


Figure 3.2: JNI General Work Flow

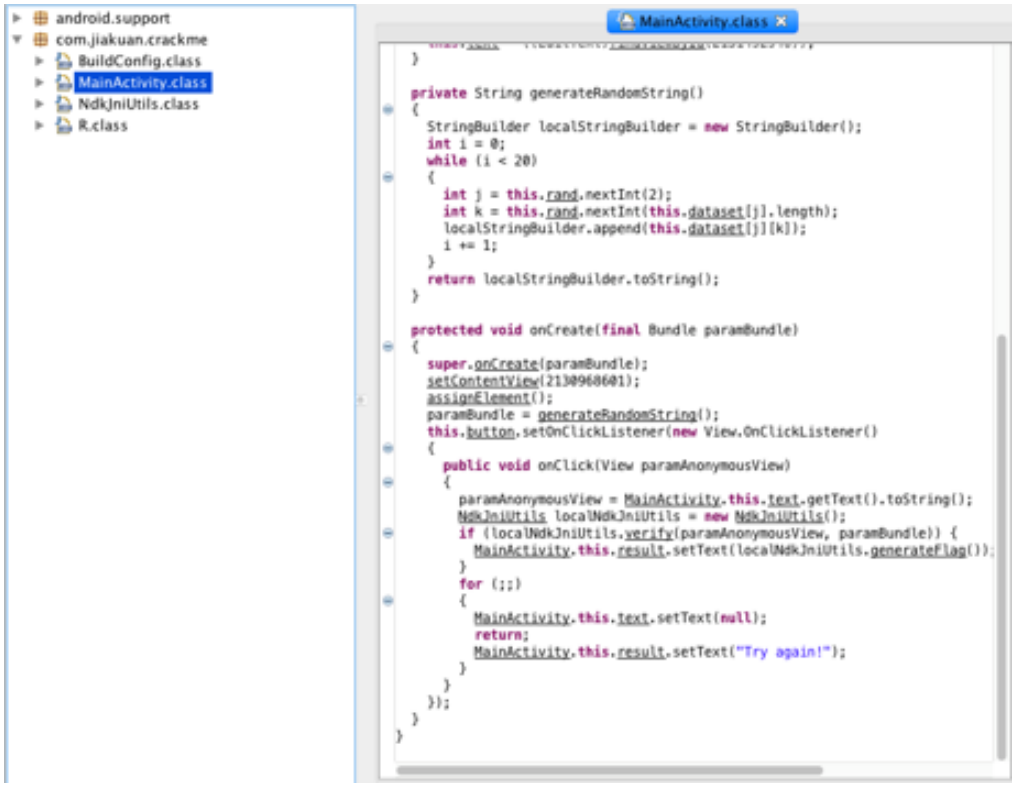


Figure 3.3: Decompiled Code for Unpacked Android Applications

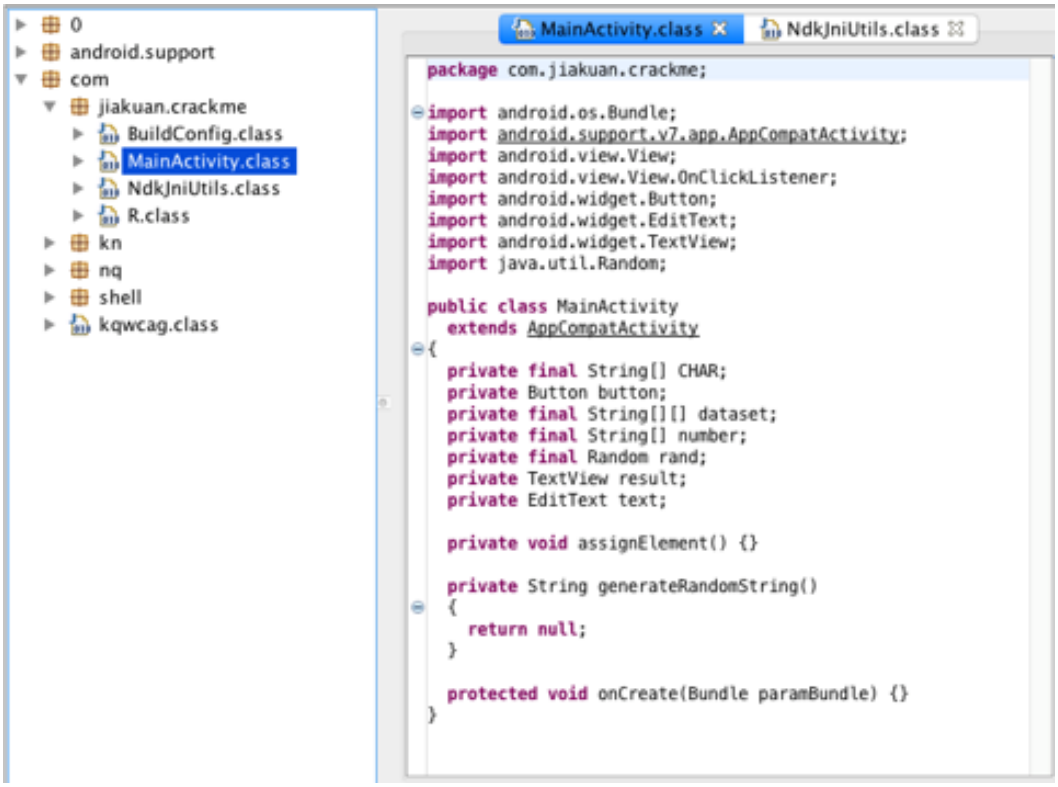


Figure 3.4: Decompiled Code for Packed Android Applications

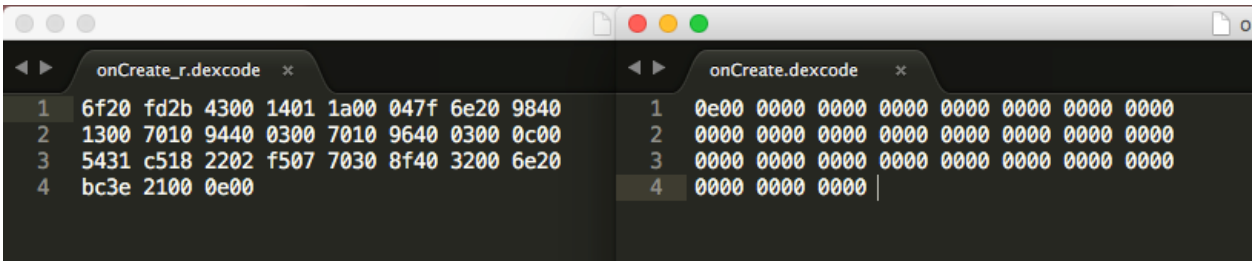


Figure 3.5: Comparison of Unpacked and Packed Application for Dumping DEX Code Content from Memory

```
public native void a1(byte[] paramArrayOfByte1, byte[] paramArrayOfByte2);
public native void at1(Application paramApplication, Context paramContext);
public native void at2(Application paramApplication, Context paramContext);
public native void c1(Object paramObject1, Object paramObject2);
public native void c2(Object paramObject1, Object paramObject2);
public native Object c3(Object paramObject1, Object paramObject2);
public native void jniCheckRawDexAvailable();
public native boolean jniGetRawDexAvailable();
public native void r1(byte[] paramArrayOfByte1, byte[] paramArrayOfByte2);
public native void r2(byte[] paramArrayOfByte1, byte[] paramArrayOfByte2, byte[] paramArrayOfByte3);
public native ClassLoader rc1(Context paramContext);
public native void s1(Object paramObject1, Object paramObject2, Object paramObject3);
public native Object set1(Activity paramActivity, ClassLoader paramClassLoader);
public native Object set2(Application paramApplication1, Application paramApplication2, ClassLoader paramClassLoader, Context paramContext);
public native void set3(Application paramApplication);
public native void set3(Object paramObject1, Object paramObject2);
public native void set4();
public native void set5(ContentProvider paramContentProvider);
public native void set8();
}
```

Figure 3.6: Code Obfuscation for Native Methods

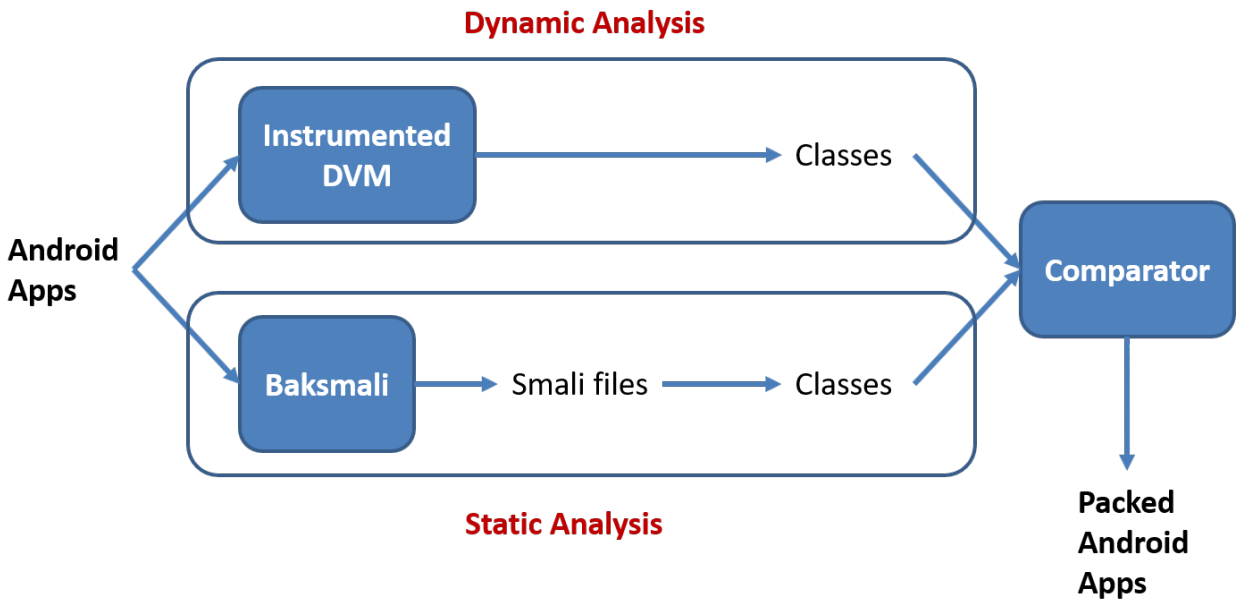


Figure 3.7: An Overview of Automatic Detecting and Unpacking for Packed Android Applications

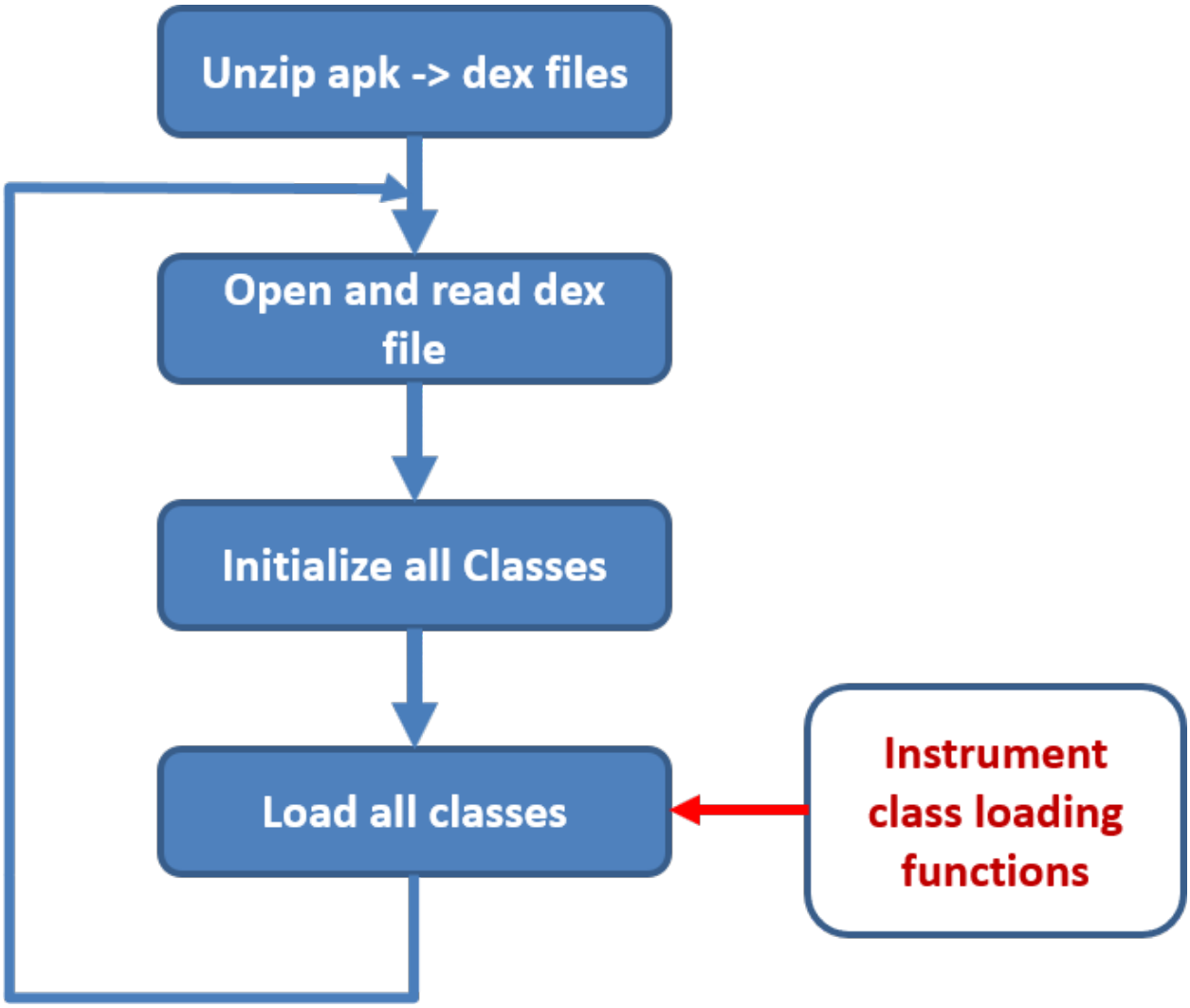


Figure 3.8: Android Runtime Class Loading

Chapter 4

MobileFindr: Function Similarity Identification for Reversing Mobile Binaries

4.1 Introduction

With the general availability of closed-source applications, there is a need to identify function similarity among binary executables. For instance, in the automatic patch-based exploit generation, detecting the function similarity/difference between a pre-patch binary and post-patch binary reveals the patched vulnerability [31, 32, 33, 34], and such information can be explored automatically within a few minutes [35], and generate 1-day exploits [36]. Performing function similarity measurement between intellectual property protected software binaries and suspicious binaries indicate potential cases of software plagiarism [37, 38, 39, 18, 40]. Detecting similar malicious functionality between different binary malware samples is another appealing application emerged in malware analysis, since the majority of malware samples are not brand new program but rather repacks or evolutions of previous known malicious function code [41, 42].

An inherent challenge shared by the above applications is the absence of source code. Binary executable becomes the only available resource to be analyzed. A number of semantics-aware binary differencing or function similarity detecting methods have been proposed. One category is to use static analysis, which is usually based on control-flow graph (CFG) comparison [31, 33, 34, 43]. At a high level, the CFG based approach extracts various robust features for a node in the control flow graph [31, 33], or learns higher-level numeric feature representations from the control flow graph [34], or converts the control flow graph into embeddings [43], then perform similarity searching for the target functions.

Although these studies have demonstrated that CFG based methods can be effective and scalable, all of these methods exclude obfuscated binaries, which appeared in a large number of mobile applications. Basic block semantics modeling is another approach for similarity measurement [44, 40, 32]. At a high level, it represents the input-output relations of a basic block as a set of symbolic formulas, which are later proved by a constraint solver for equivalence. However, the SAT/SMT solvers which are often used to measure semantic similarity are computationally expensive and impractical for large code bases of many real world mobile applications [33].

Another category relies on dynamic analysis, which is usually based on runtime execution behavior comparison. For example, previous work by Ming et al. achieves this by collecting system or API calls to slice out corresponding code segments and then check their equivalence with symbolic execution and constraint solving [42]. However, their trace logging component is an emulator based system, which cannot handle the environment-sensitive mobile applications that can detect sandbox environment. Egele et al. built a system called BLEX to capture the side effects of functions during execution [45]. Xu et al. built a tool called CryptoHunt to capture the specific features of cryptography functions with boolean formula [46]. All of their implementation are based on Intel's Pin framework [47], which is not work on mobile platforms generally with ARM instruction set architecture.

In this chapter, we aim at improving the state of the art by proposing *trace-based function similarity mapping*, a hybrid method to efficiently search for similar functions in mobile binaries. Regardless of the optimization and obfuscation difference, similar code must still have semantically similar execution behavior, whereas different code must behave differently [45]. Our key idea is to capture the dynamic behavior features during the execution of a function along a runtime trace. More precisely, we propose to record a variety of dynamic runtime information as dynamic behavior features via dynamic instrumentation, and use stack backtrace information to locate corresponding functions that can be represented with

these features. Then we calculate the similarity distance based on such features and return a list of similar functions ranked by the score of distance.

We have designed and implemented a system called *MobileFindr*, and evaluated it with a set of mobile examples under different obfuscation scheme combinations. Our experimental results show that our system can successfully identify fine-grained function similarities between mobile binaries, and outperform existing state-of-the-art approaches in terms of better obfuscation resilience and accuracy. Our evaluation with top-ranked real-world mobile applications also demonstrated the effectiveness of our system.

Correspondingly, our contributions in this work are:

- We have proposed a novel approach, *trace-based function similarity mapping*, to perform function similarity measurement on mobile platforms. Our key solution is to capture observable dynamic behaviors along an execution trace via dynamic instrumentation, and characterize functions with such behaviors. Our approach exhibits stronger resilience to various anti-reverse engineering techniques for mobile applications. To best of our knowledge, this is the first work having such ability on mobile platforms.

- We have proposed a variety of dynamic features to record during the function execution, which allow us to approximate the semantics of a function without relying on the source code access.

- we have implemented a system called *MobileFindr* and source code is publicly available at GitHub: https://github.com/tigerlyb/MobileFindr.

- We have demonstrated the viability of our approach for top-ranked real-world mobile frameworks and applications.

The rest of this chapter is organized as following. Section 4.2 introduces background and challenges. Section 4.3 presents the details of our system design and implementation. Section 4.4 presents our evaluation and results. Discussion and limitations are presented in Section 4.5. Then we present related work in Section 4.6, and conclude this chapter in Section 4.7.

4.2 Background

This section introduces the background of reverse engineering, presents the popular tools that help for reverse engineering mobile applications, including various debuggers, disassemblers, decompilers, etc. Then we demonstrate motivating examples and describe possible reverse engineering challenges that can affect the state of the art function identification methods.

4.2.1 Reverse Engineering Mobile Applications

Reverse engineering is the process of taking a program's binary code and recreating it so as to trace it back to the original source code. It is being widely used in computer software security to enhance product features without knowing the source: find security flaws, test code compatibility, add new features or redesign the product, understand the design of malicious code, etc. In this section, we present popular reverse engineering tools for mobile applications as follows:

- **Debugger**: helps developer to understand how the program behaves at runtime without modifying the code, and allows the user to view and change the running state of a program. With the release of Xcode 5, the LLDB debugger [8], which is part of the LLVM compiler development suite, becomes the foundation for the debugging experience on Apple platforms. LLDB is fully integrated with Xcode and provides deep capabilities in a user-friendly environment. For Android platform, both LLDB and JDB (Java debugger) are integrated in the Android Studio debugger [9]. By default, Android Studio automatically choose the best option for the code you are debugging. For example, if you have any C or C++ code in the project, Android studio debugger select LLDB to debug your code. Otherwise, Android Studio uses the Java debug type.

- **Disassembler**: a software tool which transforms binary code into a human readable mnemonic representation called assembly language. Many disassemblers are available on the market, both free and commercial. Apktool [2] and Baksmali [3] are free tools

that can disassemble the DEX format used by Dalvik, Android's Java VM implementation. They can decode resources to nearly original form and rebuild them after making some modifications. They also makes working with an application easier because of the project like file structure and automation of some repetitive tasks like building Android APK files, etc. The most powerful commercial disassembler is IDA Pro [6], published by Hex-Rays. It can handle binary code for a huge number of processors and has open architecture that allows developers to write add-on analytic modules.

- **Decompiler**: a software tool used to revert the process of compilation. Decompilers are different from disassemblers in one very important aspect. While both generate human readable text, decompilers generate much higher level text, which is more concise and much easier to read. For example, Android developer can use Dex2jar [4] to convert DEX file to Java class file, and then open it in JD-GUI [5] to display Java source code. Hex-Rays Decompiler [7] is a IDA Pro extension that converts native processor code into human readable C-like pseudo-code text.

4.2.2 Challenges

The software security community relies on such reverse engineering tools to analyze and validate programs. However, various anti-reverse engineering techniques employed by the latest mobile applications make existing reverse engineering tools ineffective. For instance, the anti-debugging and anti-emulator techniques employed by mobile applications limit the usage of many dynamic analysis tools [48, 49, 50]. Code obfuscation scheme provide strong protection against automated static reverse engineering tools. Moreover, different mobile applications tend to use different obfuscation techniques and even same application changes obfuscation options when updating its version. In this chapter, we focus on analyzing iOS applications. Nowadays iOS developers heavily rely on code obfuscation to evade detection since iOS is a close-source platform. Therefore, in this section, we introduce different code obfuscation features as well as motivating examples for understanding each features.

Code Obfuscation

Obfuscation aims at creating obfuscated code that is difficult for humans to understand. Obfuscation techniques include modifying names of classes, fields, and methods, reordering control flow graphs, encrypting constant strings, inserting junk code, etc. To obfuscate mobile applications, we rely on a state-of-the-art open-source obfuscation tool, Obfuscator-LLVM 4.0 [51], which supports popular obfuscation transformations as follows.

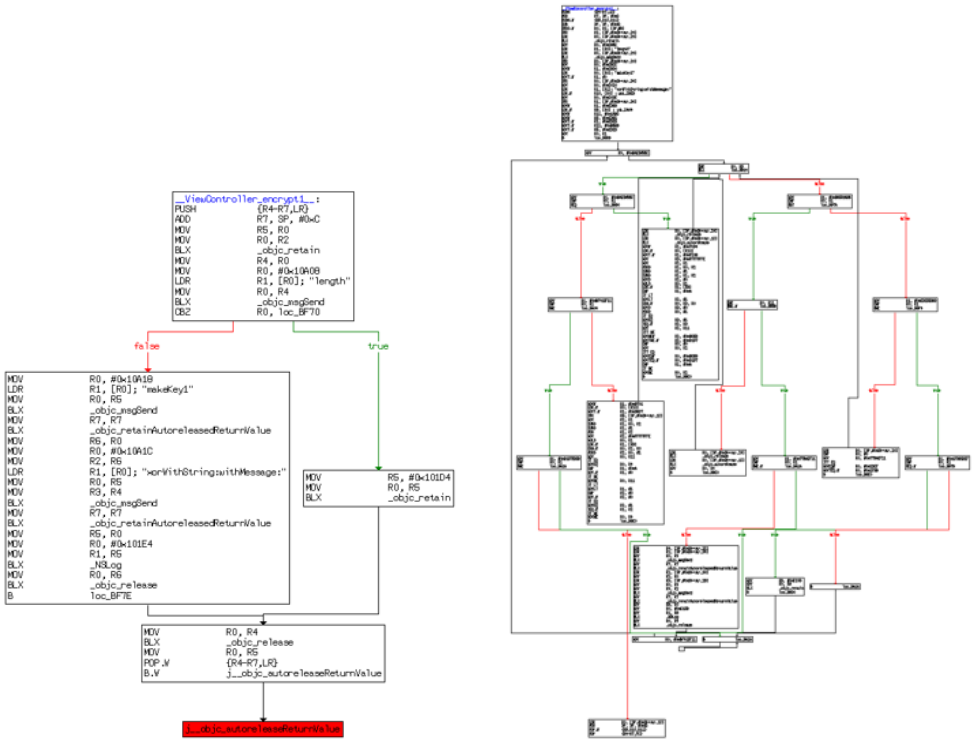
- **Control Flow Flattening**: The purpose of this pass is to completely flatten the control flow graph of a program. The flag option *-split* activates basic block splitting, which improve the flattening when applied together.
- **Instructions Substitution**: The goal of this obfuscation technique simply consists in replacing standard binary operators (like addition, subtraction or boolean operators) by functionally equivalent, but more complicated sequences of instructions.
- **Bogus Control Flow**: This method modifies a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block. The original basic block is also cloned and filled up with junk instructions chosen at random.

Obfuscation Example

We use the example in Figure 4.2 to illustrate code obfuscation on iOS platform. Figure 4.1 shows the Objective-C source code of a function called *encrypt1*. It takes a string message as input and xor the message with a key, then return the encrypted message. Figure 4.2a shows the original control flow graph without any obfuscation, which only contains 4 basic blocks. While Figure 4.2b is the obfuscated version (combined all three obfuscation options above) of that function. As mentioned in Section 4.1, existing static approaches that rely on control flow graph similarity and basic block level comparison will likely not be able to make a meaningful distinction in this scenario. Alternative approaches, such as dynamic

```
1.  - (NSString *)encrypt1:(NSString *)message {
2.      if ([message length] == 0) {
3.          return @"NULL";
4.      }
5.
6.      NSString *key = [self makeKey1];
7.      NSString *encryptedMsg = [self xorWithString:key
    withMessage:message];
8.
9.      NSLog(@"encrypt1: %@", encryptedMsg);
10.     return encryptedMsg;
11. }
```

Figure 4.1: A Motivating Example: Code



(a) (b)

Figure 4.2: A Motivating Example: CFG

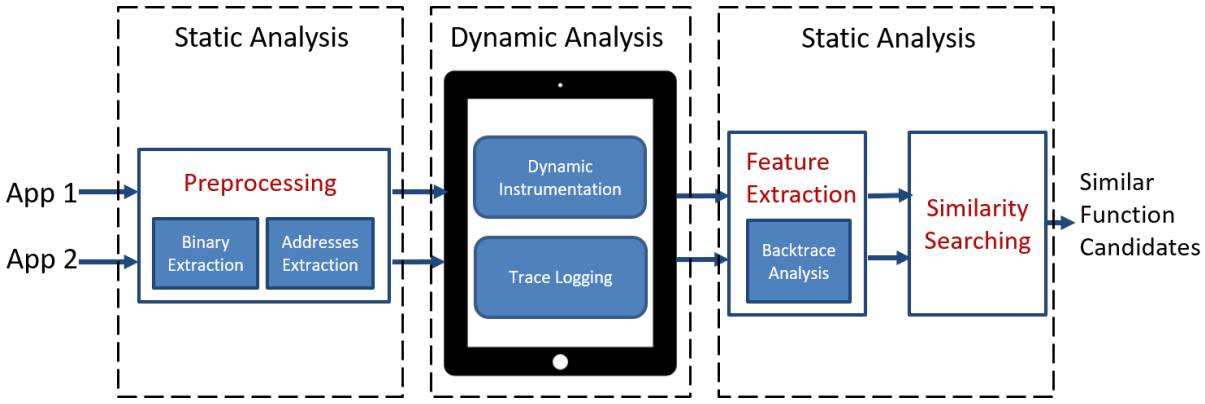


Figure 4.3: Schematic Overview of Trace-based Function Similarity Mapping System

approaches, either rely on Pin tool or emulator-based system to capture execution behavior. Pin tool is not able to work on analyzing most mobile applications, since ARM processors dominate mobile platforms. The anti-emulator techniques employed by mobile applications also limit the usage of such emulator-based analysis system. To address the above mentioned challenges in the scope of matching function for mobile binaries, we design a novel on-device dynamic instrumentation system.

4.3 Design and Implementation

In this section, we first illustrate the design of our approach, and then detail the implementation of our system.

4.3.1 Overview

We present *trace-based function similarity mapping*, a hybrid method to efficiently search for similar functions in mobile binaries. More precisely, we propose to record a variety of dynamic behavior features during the execution of a function along an execution trace. We define the concept of "dynamic behavior features" broadly to include any information that can be derived from observations made during execution. Our approach works as the

following: given two mobile applications A, B and a function of interest F from A. Both F and any executed functions from B are characterized with dynamic behavior features. Then we compute similarity scores between F and each function f from B, to identify which functions in B are similar to F. The novelty of our approach lies in the follows.

- What features are useful for semantic similarity comparisons?
- How these features are captured on mobile platforms?
- How to characterize a function with such features?

Figure 4.3 illustrates the architecture of our system, which comprises four stages: preprocessing, on-device dynamic analysis, feature extraction and similarity searching. The preprocessing stage, as shown in the left side of Figure 4.3, involves two parts: binary extraction and address extraction. It dumps the mobiles binaries from the application and extract addresses for all functions and imported libraries and frameworks. All the extracted addresses are passed to the on-device dynamic analysis stage for instrumentation and trace logging usage. The recorded traces will be analyzed by the feature extraction stage. Then we perform the similarity searching based on the function features obtained from feature extraction stage. Next, we will present each step of our system in the following sections.

4.3.2 Preprocessing

Binary Extraction

When you download an iOS application from the iOS App Store, Apple injects a special 4196 byte long header into the signed binary encrypted with the public key associated with your iTunes account. For this step we choose Clutch [52], to decrypt and dump application binary. Then we need to disable the ASLR (Address Space Layout Randomization) to get the correct function addresses.

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution

Figure 4.4: Partial vs Full ASLR in iOS

Address Space Layout Randomization is an important exploit mitigation technique introduced in iOS 4.3 [53]. ASLR makes the remote exploitation of memory corruption vulnerabilities significantly more difficult by randomizing the application objects location in the memory. By default iOS applications uses limited ASLR and only randomizes part of the objects in the memory.

In order to take full advantage of the ASLR, the application has to compiled with *-pie* flag (Generate Position-Dependent Code). This flag is automatically checked by default in the latest version of *Xcode* (from iOS 6). So, all the applications that are compiled in the latest SDK will automatically use full ASLR. Figure 4.4 compares the different memory sections for partial and full ASLR applications.

During our instrumentation process, we first need to extract the imported library addresses via IDA pro statically since our dynamic instrumentation tools doesn't provide the API to extract the library. ASLR will randomize the objects' addresses in the memory

during the execution that affect our instrumentation. We leverage the tool *removePIE* [54] to disable the ASLR by flipping the PIE flag. After that, we put the binary back to the application and re-sign it with *ldid* [55].

Address Extraction

We utilize IDA Pro [6] to disassemble the binary obtained from previous step, extract function addresses as well as imported library addresses and framework addresses through IDAPython API. This component is implemented with 155 lines of Python code. Listing 4.1 shows an example of a function address table extracted from the iOS application binary. Each line consists of starting address (e.g., 0x11834), ending address (e.g., 0x11980) and function name (e.g, *prepareToRecord* from the class *MovieRecorder*). Listing 4.2 shows an example of library addresses, which only consist the starting addresses and library names.

Listing 4.1: Function Addresses

```
...
0xb7ea,0xb964,-[VideoSnakeViewController toggleRecording:]
0xe2cc,0xe51c,-[VideoSnakeSessionManager startRecording]
0x111d8,0x1128c,-[MovieRecorder initWithURL:]
0x1161c,0x116a8,-[MovieRecorder delegate]
0x11834,0x11980,-[MovieRecorder prepareToRecord]
0x11d48,0x11ebc,-[MovieRecorder finishRecording]
...
```

Listing 4.2: Library Addresses

```
...
0x1606c,__Block_copy
0x1607c,__Block_object_assign
0x1608c,__Block_object_dispose
0x1609c,__Unwind_SjLj_Register
0x160ac,__Unwind_SjLj_Resume
0x160bc,__Unwind_SjLj_Unregister
...
```

4.3.3 On-device Dynamic Analysis

The on-device dynamic analysis stage performs dynamic instrumentation and trace logging in order to record the needed information.

Dynamic Instrumentation

We utilize Frida [56], a dynamic instrumentation toolkit, to inject JavaScripts in application process that monitor the dynamic behavior during execution. Frida lets you inject snippets of JavaScript or your own library into native applications. Frida's core is written in C and

injects Googles V8 engine into the target processes, where the JavaScript gets executed with full access to memory, hooking functions and even calling native functions inside the process.

Trace Logging

In our implementation we chose features that capture a variety of system level information (e.g., libc calls), as well as higher level attributes, such as objc calls, framework API invocations etc. Generally, systems provide a library or API that sits between normal programs and the operating system. On Unix-like systems, that API is usually part of an implementation of the C library (libc), that provides wrapper functions for the system calls. iOS is based on Unix. All the system calls are defined in *libsystem_kernel.dylib*. Tracing all system calls will result significant performance issue which makes the application stuck. Therefore, instrumenting all imported library calls as well as framework API calls meet our needs for trace logging.

- **Library Calls**: e.g., *memset*, *memcpy*, *free*, etc. defined in *libSystem.B.dylib*, *_objc_getClass*, *_objc_getProtocol*, etc. defined in *libobjc.A.dylib*. The imported libraries can be extract statically by use IDA pro. First, we extract all imported library modules with IDA API *get_import_module_qty()*, then we extract all imported library call addresses by enumerate all functions with IDA API *enum_import_name()*. After that, we use Frida's API called *Interceptor.attach(address, callback)* to inject JavaScript in the corresponding addresses and trace all the imported library calls.

- **Framework APIs**: e.g., *OpenGLES*, *CoreMedia*, *UIKit*, etc. we use Frida's API *Process.enumerateModules* to extract all framework modules loaded by the application during runtime, and then use *Module.enumerateImports(name, callbacks)* to inject JavaScript so that we can trace all the imported framework APIs by passing their names to it.

We leverage the Frida APIs to inject JavaScript at the library addresses and framework addresses to record the invocations of such features above, and generate a backtrace for the current thread, returned as an array of native pointer addresses for the subsequent steps.

4.3.4 Feature Extraction

Listing 4.3 illustrates the logged trace data, which consists of arrays of addresses. Each line indicates an invocation of library call or framework API call, followed by its stack backtrace information. First, we transform the addresses to function names according to the address table obtained from the preprocessing stage. For instance, 0x1609c is the starting address of *__Unwind_SjLj_Register*, 0x11892 is in the range of 0x11834 and 0x11980, which indicate the library *__Unwind_SjLj_Register* is called by function *prepareToRecord*. The rest can be done in the same manner. Listing 4.4 illustrates a full translated results from Listing 4.3.

Listing 4.3: Stack Backtrace: Address

```
...
0x1609c,0x11892,0xe498,0xb92e,0xb15a
0x1621c,0x118c0,0xe498,0xb92e,0xb15a
0x1620c,0x118fc,0xe498,0xb15a
...
```

Listing 4.4: Stack Backtrace: Name

```
...
__Unwind_SjLj_Register,-[MovieRecorder prepareToRecord],-[
   VideoSnakeSessionManager startRecording],-[
   VideoSnakeViewController toggleRecording:],sub_B120
_dispatch_get_global_queue,-[MovieRecorder prepareToRecord],-[
   VideoSnakeSessionManager startRecording],-[
   VideoSnakeViewController toggleRecording:],sub_B120
_dispatch_async,-[MovieRecorder prepareToRecord],-[
   VideoSnakeSessionManager startRecording],-[
   VideoSnakeViewController toggleRecording:],sub_B120
...
```

Next, we match these library calls or framework API calls to its corresponding caller functions as features. Listing 4.5 represents features of function *prepareToRecord*, in JSON format. The feature extraction component is implemented with 280 lines of Python code.

4.3.5 Similarity Searching

The function feature representation is a length-N feature list. We chose Jaccard index to measure the similarity between lists. The Jaccard similarity index (Jaccard similarity coefficient) compares members for two sets to see which members are shared and which are distinct. Its a measure of similarity for the two sets of data, with a range from 0% to 100%. The formula to find the index is:

$$J(X,Y) = |X \cap Y|/|X \cup Y| \tag{4.1}$$

We define sim(f, g) to be the similarity score between function f and g. We perform similarity searching as the following: starting with a known reference function in a trace, we

are searching for mobile binaries containing similar functions by calculating similarity score and listing top K similar function candidates.

Listing 4.5: Function Features

```
{
        "name" : "-[MovieRecorder prepareToRecord]",
        "features" : [
                [
                        "__Unwind_SjLj_Register",
                        "_dispatch_get_global_queue",
                        "_dispatch_async",
                        "__Block_object_assign",
                        "__Unwind_SjLj_Unregister"
                ]
        ]
}
```

4.4 Evaluation

In this section, we evaluate our system to answer the following questions.

- **Q1**: Can MobileFindr detect similar functions in different versions of same mobile applications?
- **Q2**: Can MobileFindr detect similar functions used by different mobile applications?
- **Q3**: Can MobileFindr be used for analyzing real-world mobile applications?

Particularly, we conduct our experiments to evaluate whether our system outperforms existing binary similarity detection tools in terms of better obfuscation resilience and accuracy. We designed two controlled datasets so that we have a ground truth to assess comparison

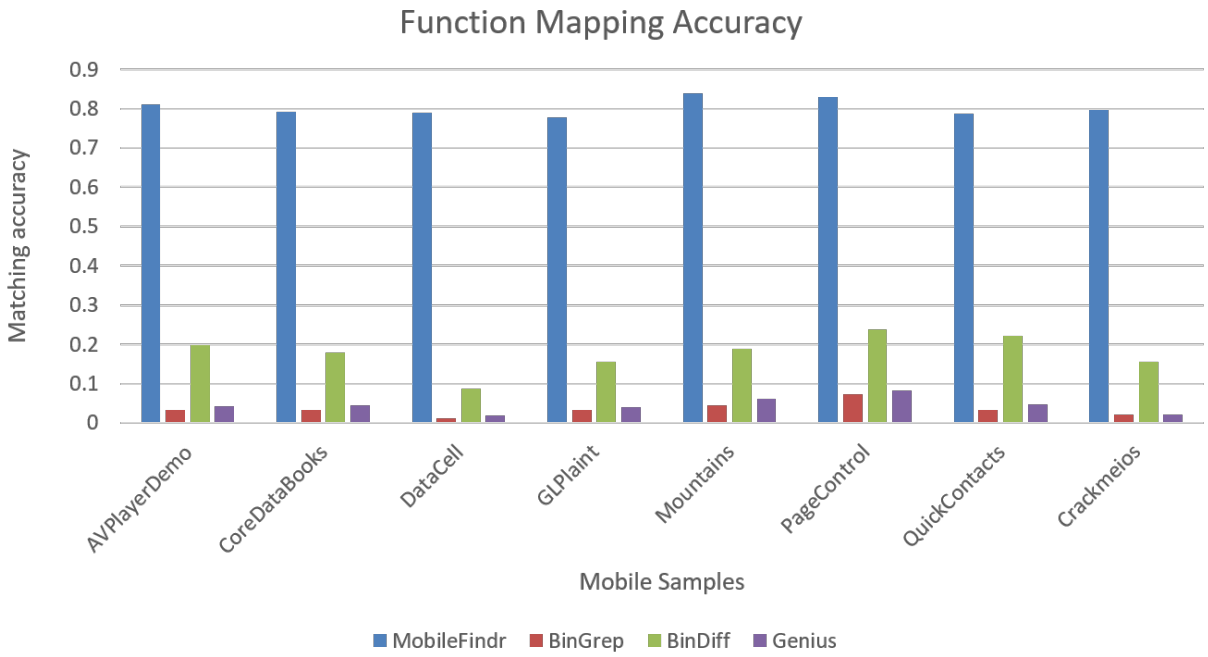


Figure 4.5: Function Mapping between Obfuscated Version and Non-obfuscated Version

results accurately. We also evaluate the effectiveness of our system in analyzing real world top-ranked iOS applications from Apple App Store.

4.4.1 Experiment Setup

Our on-device dynamic analysis is performed on a 32GB Apple Jailbroken iPad (4th Generation) running iOS 8.3. The configuration of our testbed machine for feature extraction and similarity searching is shown as follows.

- CPU: Intel Core i7-6700K Processor (Eight-core with 4.00GHz)
- Memory: 64GB
- OS: Ubuntu Linux 14.04 LTS
- Python Version: 2.7.12

- IDA Pro Version: 6.6

4.4.2 Ground Truth Dataset

Data 1

First, we collect 8 sample codes with different functionalities from official Apple developer website. For each sample we build both non-obfuscated version and obfuscated version. The obfuscated version combines all three settings in Table 4.1.

Data 2

Then we test our system with third-party frameworks or libraries that are commonly used by popular mobile applications. In practice, programmers usually take advantage of existing frameworks or libraries to speed up their developments. In our evaluation, we choose *AFNetworking* and *SDWebImage*, top-two ranked open source frameworks [57] as the reference implementation.

AFNetworking [58] is an Objective-C networking library for iOS, macOS and tvOS. It is a robust library that has been around for many years. From basic networking to advanced features such as Network Reachability and SSL Pinning, *AFNetworking* has it all. It is one of the most popular iOS libraries of all time with almost 50 million downloads.

SDWebImage [59] is an asynchronous image downloader with caching. It has handy UIKit categories to do things such as set a UIImageView image to an URL. While networking has become a little bit easier in Cocoa over the years, the basic task of setting an image view to an image using an URL hasnt improved much. *SDWebImage* helps ease a lot of pain, so thats why its so popular with iOS application developers.

Our purpose is to detect such frameworks or libraries that commonly used in different mobile applications. To this end, we collect 8 open source projects from GitHub, and reuse the provided APIs from two libraries above. We built sample applications with non-obfuscated

version and 7 different combinations of the obfuscation settings, which results in 64 applications in 8 different types. We kept the debug symbols as they provide a ground truth and enable us to verify the correctness of matching using the functions symbolic names.

Table 4.1: Different Obfuscation Types and Flag Settings

	Type	Flag Setting
1	control flow flattening	-fla, -split, -split_num=3
2	instruction substitution	-sub, -sub_loop=3
3	bogus control flow	-bcf, -bcf_loop=3, -bcf_prob=40

4.4.3 Obfuscation Options

As mentioned in section 4.2, we use Obfuscator-LLVM to obfuscate our ground truth mobile samples. Table 4.1 lists specific obfuscation settings that we use to build our ground truth iOS samples. We integrate Obfuscator-LLVM into Xcode, and enable the three obfuscation features described in Section 4.2, and apply different settings as follows.

- Control Flow Flattening: The flag *-fla* activates control flow flattening. The flag *-split* activates basic block splitting, which improve the flattening when applied together. If *-split_num=3*, applies it 3 times on each basic block.

- Instructions Substitution: The flag *-sub* activates instructions substitution. If *-sub_loop=3*, applies it 3 times on a function.

- Bogus Control Flow: The flag *-bcf* activates the bogus control flow pass. If *-bcf_loop=3*, applies it 3 times on a function. Default number is 1. If The option *-bcf_prob=40*, a basic block will be obfuscated with a probability of 40%.

4.4.4 Peer Tools

We compare our tools with other state-of-the-art similarity detection or diffing tools that open to public: BinDiff, BinGrap, Genies. BinDiff [60] is a comparison tool for binary files,

that assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code. BinGrap [61] is also a static analysis tool that perform function similarity searching, but it can output a list of functions in order of similarity. Genius is a bug search engine that performs function similarity matching based on mapping raw features of a function into a higher-level numeric vector where each dimension of the vector is the similarity distance to a categorization in the codebook. However, only partial code is available, including raw feature extraction and search. Therefore, we re-implement Genius' two core steps, codebook generation and feature encoding in python. We utilized Hungarian algorithm for calculating bipartite graph matching cost and normalized spectral clustering [62] for ACFGs (Attributed Control Flow Graph) clustering. In evaluation phrase, we adopt Nearpy [63] for LSH (Locality Sensitive Hashing) [64] and search. We used SQLite to store function information and encoded vectors.

As mentioned in section 4.1, BLEX [45], BinSim [42] and CryptoHunt [46] don't work on mobile platforms. To the best of our knowledge, we are the first to propose a dynamic strategy for comparing mobile binary code. This is the reason why we did not compare our evaluation to these dynamic approaches.

4.4.5 Evaluation Results

We use the recall rate (A.K.A true positive rate) mentioned in [34] as the evaluation metrics to evaluate the accuracy of the following methods. In the code search scenario, the search results are a ranked list. For each query q, there are m matching functions out of a total of L functions. If we consider the top-K retrieved instances as positives, the total number of correctly matched functions, μ, are true positives, and the remaining number of functions in the top K, that is K - μ, are false positives. Based on the definition, the recall rate is calculated as $recall(q) = \mu \div m$.

The first evaluation method is to answer **Q1**. Dataset 1 is used for this evaluation. For each sample, We randomly select functions from non-obfuscated version as reference functions,

then perform our *trace-based function similarity mapping* to see if we can locate the same function in obfuscated version listed as top-K similar function candidates. Figure 4.5. shows the comparison results between BinGrep, BinDiff, and Genius for different applications. For example, MobileFindr ranks 82.9% functions at top 1 for PageControl App, whereas Genius only ranks 8.2%.

The second round of evaluation method worked on the dataset 2, which is to answer **Q2**. Both functions in dataset 1 and 2 have known ground truth for metric validation. We randomly select one application from each type of applications as reference known application, and select functions in *AFNetworking* and *SDWebImage* from that application as query functions. Then we perform *trace-based function similarity mapping* for searching the given functions in the rest applications, and list top K candidates for each application based on the similarity score. We only compare with Genius and BinGrep since BinDiff is a one-to-one mapping tool, which cannot list more than 1 candidate. data 2 is shown in Figure 4.6.

Our evaluation results show that MobileFindr can achieve more than 80% accuracy in average from top 3 to top 15 similar functions, which outperforms other tools in terms of much more better accuracy and obfuscation resilience.

4.4.6 Real-world Application Case Study

We conduct third evaluation to answer **Q3**. This section gives an empirical evaluation to test MobileFindr using real-world applications to evaluate its efficiency. We evaluated 6 top-ranked iOS applications in different types, such as search engine, social networking, etc. For instance, Baidu is the world's largest Chinese search engine. We downloaded two different versions of Baidu application, version 930 and version 935. We chose version 930 as reference application and performed a simple web searching with key words: *security* for trace logging. We collected 430 functions in this trace, and then perform *trace-based function similarity mapping* to search similarity functions in the new version 935, and listed top 10 similar function candidates. We manually verified the matched functions by looking at their

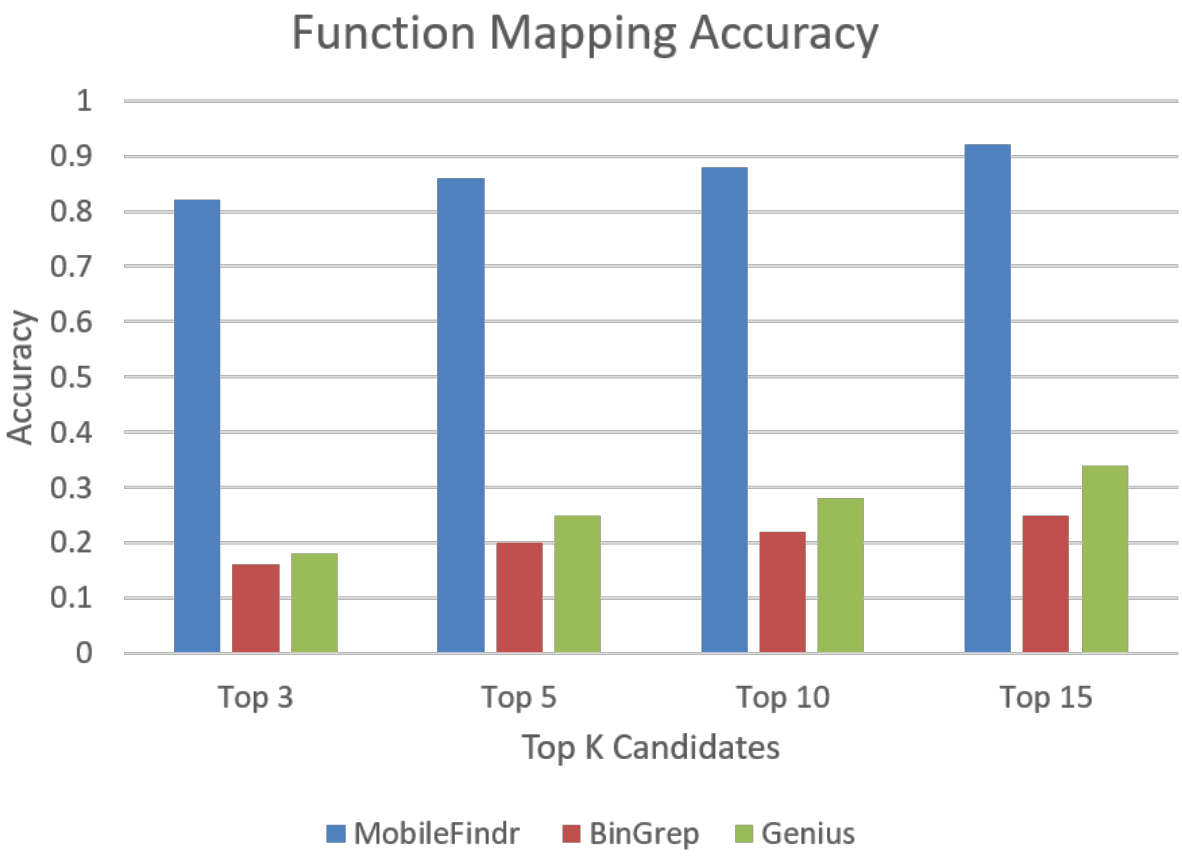


Figure 4.6: Function Mapping Evaluation for Popular Third-party Frameworks

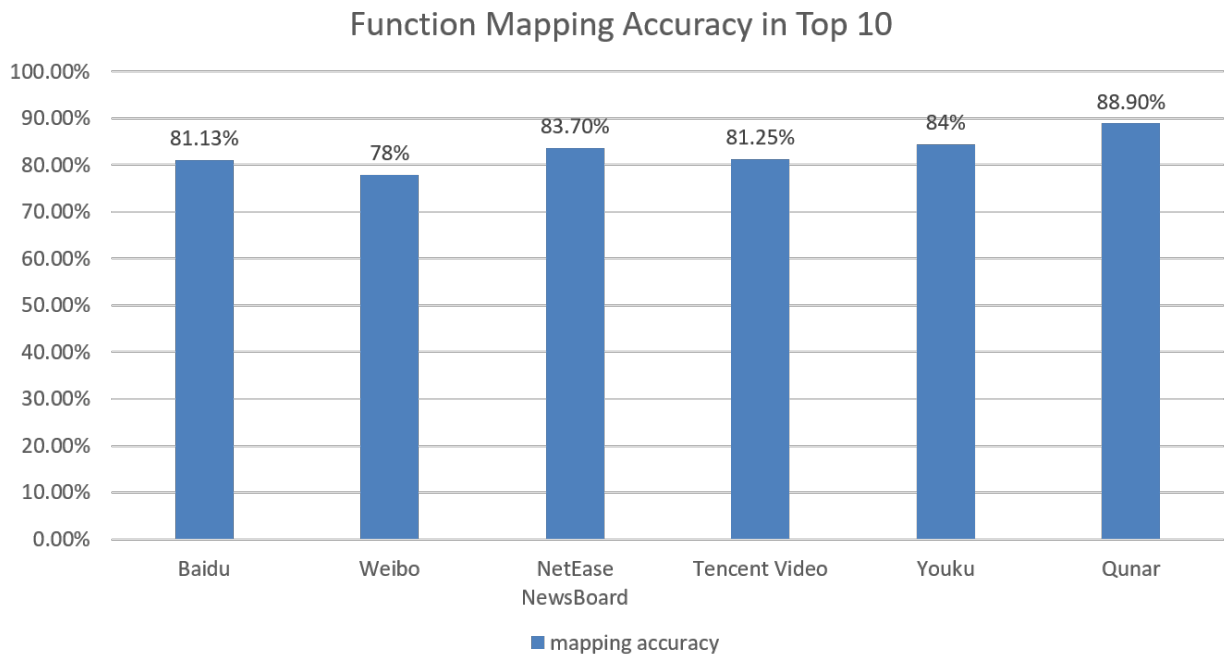


Figure 4.7: Function Mapping Evaluation in Real-world Applications

assembly code and compare the code logic. MobileFindr achieve 81.13% accuracy with less than 10 minutes for the matching process. While matching the same 430 functions in Genius, it only achieved 59.7% accuracy, but spent around 2 hours in training, more than 40 hours when handling function graph embeddings. Figure 4.7 shows the function mapping results for the 6 real-world applications.

4.5 Discussion

In this section, we discuss the limitations of our system and potential solutions to be investigated in future work.

First, a challenge that we already touched upon in Section 4.4 is the fact that our approach needs manual verification efforts for real world iOS applications, since we don't have access to their source code. The candidate similarity ranking produced by our system gives an

ordered list of matched functions that have to be manually inspected by an analyst to verify if those functions are actually similar. Some of the existing dynamic approaches [42, 46] rely on symbolic execution to generate a set of symbolic formula, and then use theorem prover to perform the equivalence checking. However, the theorem prover is computationally expensive and impractical for large code bases of many real world mobile applications. Such an automatic verification would be ideal, but surely is a research topic in itself and is outside the scope of this work.

Second, the incomplete path coverage is a concern for all dynamic analysis system, including ours. The possible solutions are to explore more paths by automatic input generation [65, 66]. To trigger as many dynamic behaviors as possible for trace logging, we can leverage the idea of Malton [25], which proposed an efficient path exploration technique that employs in-memory concolic execution with an offloading mechanism and direct execution engine.

Third, the functions considered by us need to have a certain amount of complexity for the approach to work effectively. Otherwise, the relatively low combination number of library calls leads to a high probability for collision. Hence, we only considered functions with at lease five basic blocks, as noted in Section 4.4. For instance, the potential for bugs in small functions, however, is significantly lower than in large functions, as shown in [67]. Hence, in a real-world scenario this should be no factual limitation.

4.6 Related Work

There has been a substantial research on detecting binary code similarity. Existing semantics aware binary matching techniques can be classified into different categories. One is based on static information including numeric features and structural features [68, 33, 34]. Many numeric features (e.g. the number of basic blocks, the number of edges, logic instructions,local variables, etc) and control flow graph has been demonstrated to be robust across compilers and different compile options in previous work [31, 44]. Another one executes target code and

collect runtime behavior [45, 69, 46, 42]. Common execution behaviors includes stack and heap memory access, system call sequences and library calls, registers values, execution path, etc. The other one based on the basic block modeling. Cop [40] models the program semantics an three different levels: basic block, path, and whole program. To model the semantics of a basic block, they adopt the symbolic execution technique to obtain a set of symbolic formulas that represent the input-output relations of the basic block in consideration. To compare the similarity or equivalence of two basic blocks, they check via a theorem prover the pair-wise equivalence of the symbolic formulas representing the output variables, or registers and memory cells.

The combination of collected features represent as a signature of target code for matching step. It is vital to identify robust features and correctly characterize target code with the features. Bindiff [60] as an efficient binary diffing tool using a graph theoretic approach to find similarities and differences. The graph isomorphism detection on pairs of function works well when two semantically equivalent binaries have similar control flow. But CFG changes across architectures and compilers. In [34], Genius maps raw features of a function into a higher-level numeric vector where each dimension of the vector is the similarity distance to a categorization in the codebook. However, one common limitation of static approaches is incapable of handling obfuscated code. BLEX [45] collects execution side effects during function execution and uses a multidimensional vector as function signature for similarity assessment. It relies on Pin framework and can not apply to mobile binaries.

The techniques of binary matching have been driven towards to solve security problems. One common case in vulnerability assessment is that secure analysts would want to use a sample of vulnerable binary without source code to search for the similar bug across all the softwares installed in the company devices [33, 70]. It is challenging for vulnerability assessment in a large code base for the following reasons: first, most commercial software projects are closed-source and only available in the binary form without debug information. Second, different versions of software may be compiled on different optimization levels and

different compile tool-chain, which would radically changes both the number of nodes and structure of edges in both the control flow graph and the call graph. Third, pervasive code protection schemes, such as class and method rename, encryption of strings, control flow obfuscation and virtualization of code, render code analysis time consuming. Our evaluation have considered above situations and demonstrate that our approach can handle it.

One derived case of vulnerability assessment is patch analysis, which focus on identifying un-patched bug duplicates [32, 35]. Since mobile application developers tend to constantly review and test their products security, they periodically release patches for new found security bugs. However, a patch typically introduces few changes and it takes a long period for users to update a software due to unawareness of security problem. Our work can also be seen in the light of a version diffing engine to identify functions between updated version and un-patched mobile binary.

With rapid development of open-source projects, the similarity between an licensed protected binary and a suspicious binary indicates a potential case of software plagiarism [40, 38]. Existing code similarity measurement methods have been proved to be useful but remain far from perfect. Some software plagiarism detection approaches based on dynamic system call sequences have also been proposed [38, 37], but they incur false negatives when the number of system calls are insufficient or when system call replacement is applied. Most of the existing methods are not effective in the presence of obfuscation techniques. Another obfuscation resilient method [40] based on symbolic execution and theorem proving bears high computational overhead.

4.7 Conclusion

We proposed MobileFindr, an on-device trace-based function similarity mapping system for reverse engineering mobile applications. It records a variety of dynamic runtime information as dynamic behavior features via dynamic instrumentation, and use stack backtrace information to locate corresponding functions that can be represented with these features. We

evaluated it with a set of examples under different obfuscation scheme combinations. Our experimental results show that our system can successfully identify fine-grained function similarities between mobile binaries, and outperform existing state-of-the-art approaches in terms of better obfuscation resilience and accuracy. Our evaluation with top-ranked real-world frameworks and applications also demonstrated the effectiveness of our system. To the best of our knowledge, we are the first to propose a dynamic strategy for function similarity identification on the mobile platform, which is capable of mitigating many anti-reverse engineering techniques.

Chapter 5

Summary

Designing a system that collecting, organizing, and evaluating facts about a mobile application and the environment in which it operates is an effective way for automating reverse engineering analysis and fight against anti-reverse engineering techniques on mobile platforms. In this dissertation, we introduce state-of-the-art reverse engineering techniques as well as different anti-reverse engineering techniques employed by mobile applications. We discuss various reverse engineering challenges and limitations of existing research studies, and present novel system techniques for reversing mobile applications on different mobile platforms (Android and iOS).

On the Android platform, Android packing services provide significant benefits in code protection by hiding original executable code, which help application developers to protect their code against reverse engineering. In Chapter 3, we investigate a series of Android packing services appeared recently, study the packing techniques adopted by packers and the difference between packed applications and unpacked applications, and presents an automatic analysis system for packed Android applications. More specifically, we present a novel system that provides a comprehensive view of packed Android applications behavior by conducing both byte-code level and native code level monitoring and information flow tracking via Android source code instrumentation. The byte-code level analysis instruments Dalvik Virtual Machine (DVM) and Android Runtime (ART) to extract hidden class information during the applications execution, and then reassemble the original DEX files that was hiding by the packer. The native code level analysis instruments the Java-Native-Interface (JNI) to monitors the execution of native components. This monitoring analysis can be used to

reveal the behavior of a packer. We have evaluated this system with open source Android applications as well as real world applications, and demonstrated the effectiveness of code detection and DEX reassemble process.

MobileFindr aims to identifying function similarity during the reverse engineering process on iOS platform. Detecting code similarity at binary level has been applied to a broad range of software security applications and reverse engineering tasks. In Chapter 4, we present a trace-based function mapping system that detects function similarity at binary level across different optimization options and obfuscation levels. Our trace-based approach captures various runtime behavior features (e.g., library call, framework API call, etc.) through multi-layer monitoring via enhanced dynamic binary instrumentation, then characterize functions with collected behaviors and perform function matching via distance calculation. We conducted experiments on real world examples, ranging from popular mobile frameworks to top-ranked mobile applications. Our experimental results show that our system can successfully identify fine-grained function similarities between iOS binaries, and outperform existing state-of-the-art approaches in terms of better obfuscation resilience and accuracy. Our evaluation with top-ranked real-world mobile applications also demonstrated the effectiveness of our system.

Bibliography

[1] Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2018-12-8.

[2] Apktool - a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/. Accessed: 2018-01-30.

[3] smali/baksmali wiki. https://github.com/JesusFreke/smali/wiki. Accessed: 2018-01-30.

[4] dex2jar. https://github.com/pxb1988/dex2jar. Accessed: 2018-01-30.

[5] Jd-gui. http://jd.benow.ca/. Accessed: 2018-01-30.

[6] Ida. https://www.hex-rays.com/products/ida/index.shtml. Accessed: 2018-01-30.

[7] Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/index.shtml. Accessed: 2018-01-30.

[8] The lldb debugger. https://lldb.llvm.org/. Accessed: 2018-01-30.

[9] Android studio - debug your app. https://developer.android.com/studio/debug/index.html. Accessed: 2018-01-30.

[10] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[11] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. Android malware detection using parallel machine learning classifiers. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 37–42. IEEE, 2014.

[12] Markus Geimer, Sameer S Shende, Allen D Malony, and Felix Wolf. A generic and configurable source-code instrumentation component. In *International Conference on Computational Science*, pages 696–705. Springer, 2009.

[13] Torsten Kempf, Kingshuk Karuri, and Lei Gao. *Software Instrumentation*, pages 1–11. American Cancer Society, 2008.

[14] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.

[15] Smartphone os market share, 2015 q2. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed: 2016-01-30.

[16] Artyom Dogtiev. App revenue statistics 2015. `http://www.businessofapps.com/app-revenue-statistics/`. Accessed: 2016-01-30.

[17] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[18] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013.

[19] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[20] Rowland Yu. Android packers: facing the challenges, building solutions. In *Proceedings of the 24th Virus Bulletin International Conference*, 2014.

[21] Avl annual android malware report 2015. `http://blog.avlyun.com/2016/02/2759/2015report/`. Accessed: 2016-01-30.

[22] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses*, pages 359–381. Springer, 2015.

[23] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Computer Security–ESORICS 2015*, pages 293–311. Springer, 2015.

[24] Yeongung Park. We can still crack you! general unpacking method for android packer (no root). *Black Hat Asia*, 2015.

[25] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *In 26th USENIX Security Symposium (USENIX Security 17). ACM*, 2017.

[26] Android on x86: Java native interface and the android native development kit. `http://www.drdobbs.com/architecture-and-design/android-on-x86-java-native-interface-and/240166271`. Accessed: 2018-01-30.

[27] F-droid. `https://f-droid.org/en/`. Accessed: 2018-01-30.

[28] Android packer analysis for real world applications. `https://docs.google.com/spreadsheets/d/1R4xqWHsbX38BwGqwqOZAus0MnX6GO5Z2eWXb2W3PxWs/edit?usp=sharing`. Accessed: 2018-01-30.

[29] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.

[30] Bo Li, Yibin Liao, and Zheng Qin. Precomputed clustering for movie recommendation system in real time. *Journal of Applied Mathematics*, 2014, 2014.

[31] Halvar Flake. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*, pages 161–174. Citeseer, 2004.

[32] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[33] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[34] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[35] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[36] Jeongwook Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. *Black Hat. Black Hat*, 2009.

[37] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM*

SIGKDD international conference on Knowledge discovery and data mining, pages 872–881. ACM, 2006.

[38] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 280–290. ACM, 2009.

[39] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Detecting software theft via system call based birthmarks. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 149–158. IEEE, 2009.

[40] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM, 2014.

[41] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 349–358. ACM, 2012.

[42] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium. USENIX Association*, pages 253–270, 2017.

[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[44] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.

[45] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX, 2014.

[46] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 921–937. IEEE, 2017.

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

[48] Dorien Herremans. Morpheus: automatic music generation with recurrent pattern constraints and tension profiles. 2016.

[49] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[50] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780. ACM, 2015.

[51] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[52] Clutch 2.0.4. https://github.com/KJCracks/Clutch/releases/tag/2.0.4. Accessed: 2018-01-30.

[53] Disable aslr on ios applications. http://www.securitylearn.net/2013/05/23/disable-aslr-on-ios-applications/. Accessed: 2018-01-30.

[54] Disable aslr on ios applications. http://www.securitylearn.net/2013/05/23/disable-aslr-on-ios-applications/. Accessed: 2018-01-30.

[55] ldid. http://iphonedevwiki.net/index.php/Ldid. Accessed: 2018-01-30.

[56] Frida. https://www.frida.re/. Accessed: 2018-01-30.

[57] Top 10 libraries for ios developers. https://www.raywenderlich.com/177482/top-10-ios-developer-libraries. Accessed: 2018-01-30.

[58] Afnetworking. https://github.com/AFNetworking/AFNetworking. Accessed: 2018-01-30.

[59] Sdwebimage. https://github.com/rs/SDWebImage. Accessed: 2018-01-30.

[60] Zynamics bindiff. https://www.zynamics.com/bindiff.html. Accessed: 2018-01-30.

[61] Bingrep. https://github.com/hada2/bingrep. Accessed: 2018-01-30.

[62] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002.

[63] Nearpy. https://github.com/pixelogik/NearPy. Accessed: 2018-01-30.

[64] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

[65] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[66] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[67] More complex = less secure: Miss a test path and you could get hacked. `http://www.mccabe.com/pdf/MoreComplexEqualsLessSecure-McCabe.pdf`. Accessed: 2018-01-30.

[68] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–94. ACM, 2017.

[69] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *ACM SIGPLAN Notices*, volume 48, pages 391–406. ACM, 2013.

[70] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

Appendix A

Android System Instrumentation Configuration

The script **analysis.sh** will generate a configure file that enables the detection and unpacking process. The configure file must be pushed into the device with the following content settings.

- **Specify the application to be analyzed**:
  - pkgname=⟨package name⟩
- **Enable class loading detection, set to true by default**:
  - enable_class_detection=true
- **Enable Java method trace, set to false by default**:
  - enable_java_method_trace=false
- **Enable libc trace, set to false by default**:
  - enable_lib_trace=false
- **Enable Java-to-Native trace, set to false by default**:
  - enable_jni_j2n_trace=false
- **Enable Java-to-Native trace for specific native method**:
  - j2n_function_name=⟨native function name⟩
- **Force change the return value of the native method**:

  - j2n_function_return_value=⟨a string value⟩

- **Force change the arg value of the native method**:
  - j2n_function_arg_index=⟨index id⟩
  - j2n_function_arg_length=⟨index length⟩
  - j2n_function_arg_value=⟨a string value⟩
- **Enable Native-to-Jave trace, set to false by default**:
  - enable_jni_n2j_trace=false
- **Force change the return value of Java method**:
  - n2j_function_name=⟨Java function name⟩
  - n2j_function_return_value=⟨a string value⟩

Appendix B

MobileFindr Usage

Step by step usage for MobileFindr:

- **Download usbmuxd-1.0.8 and create connection via USB**:
  - $ python usbmuxd-1.08/tcprelay.py -t 4444:4444 -t 22:10022
- **Download clutch 2.0.4 and copy it to the iOS device**:
  - $ scp -P 10022 Clutch-2.0.4 root@localhost:/usr/bin/Clutch
- **Copy the script "clutch_binary_dump.py" to device**:
  - $ scp -P 10022 clutch_binary_dump.py root@localhost: /
- **Connect to device vis SSH**:
  - $ ssh -p 10022 root@localhost
- **Decrypt the app binary**:
  - $ python clutch_binary_dump.py
- **Load the script preprocessing.py" into the IDA Pro to generate the function addresses file**
- **Trace logging**:
  - $ python frida_trace.py ⟨address file folder⟩ ⟨app process id⟩

- **Feature generation**:
  - $ python generate_features.py ⟨trace file folder⟩
- **Similarity Matching**:
  - $ python similarity_search.py ⟨reference app feature folder⟩ ⟨target app feature folder⟩ ⟨Top K number⟩