8-10-2018

Automating Mobile Device File Format Analysis

Richard A. Dill

Automating Mobile Device File Format Analysis

DISSERTATION

Rich Dill, Maj, USAF

AFIT-ENG-DS-18-S-008

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-DS-18-S-008

AUTOMATING MOBILE DEVICE FILE FORMAT ANALYSIS

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy in Computer Science

Rich Dill, B.S.C.S., M.S.C.

Maj, USAF

10 July 2018

AFIT-ENG-DS-18-S-008

AUTOMATING MOBILE DEVICE FILE FORMAT ANALYSIS

DISSERTATION

Rich Dill, B.S.C.S., M.S.C.
Maj, USAF

Committee Membership:

Dr. Gilbert L. Peterson, PhD, Chairman

Dr. Douglas D. Hodson, PhD, Member

Lt Col Richard S. Seymour, PhD, Member

Dr. Jack E. McCrae, PhD, Dean's Representative

Abstract

Forensic tools assist examiners in extracting evidence from application files from mobile devices. If the file format for the file of interest is known, this process is straightforward, otherwise it requires the examiner to manually reverse engineer the data structures resident in the file. This research presents the Automated Data Structure Slayer (ADSS), which automates the process to reverse engineer unknown file formats of Android applications. After statically parsing and preparing an application, ADSS dynamically runs it, injecting hooks at selected methods to uncover the data structures used to store and process data before writing to media. The resultant association between application semantics and bytes in a file reveal the structure and file format. ADSS has been successfully evaluated against Uber and Discord, both popular Android applications, and reveals the format used by the respective proprietary application files stored on the filesystem.

Acknowledgements

The Christian shoemaker does his duty not by putting little crosses on the shoes, but by making good shoes, because God is interested in good craftsmanship.

– Martin Luther

Thanks to the many people who helped make completing this dissertation possible. My wife and kids who let me exchange time with family to conduct research; my adviser (Dr. Peterson) for challenging my research and writing; my fellow students (Camdon Cady, Chris Wayne, and David King) who let me brainstorm ideas with them; the technical experts (Ben Gruverand Ole Andr V. Ravns) for answering my questions about Smali and Frida, respectively.

Rich Dill

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

Abbreviation		Page
VPFN	Virtual Page Frame Number	23
ABI	Application Binary Interface	24
BLOB	Binary Large OBject	28
DBI	Dynamic Binary Instrumentation	31
FFEx86	File-Format Extractor for x86	35
HFSM	Hierarchical Finite State Machine	35
IO	Standard Input Output	35
REWARDS	Reverse Engineering Work for Automated Revelation of Data Structures	40
SDCF	Static and Dynamic Combined Framework	42
ADB	Android Debug Bridge	57

AUTOMATING MOBILE DEVICE FILE FORMAT ANALYSIS

I. Introduction

The forensic community faces a serious problem – tools for mobile device analysis do not fully extract evidentiary data from cell phones. The leading commercial forensic tools (Oxygen, XRY, Blacklight, Cellebrite, Encase) [13] each provide differing capabilities to the forensic community, but even in summation, they are unable to extract all the available forensic data. The fault does not lie with the forensic tools or the developers, but with the changing application landscape, encompassing frequent software updates to a huge repository of available applications – 3.5M from the Google Play Store and 2.2M from the Apple Store as of Dec 2018 [14].

While changes to software bring security enhancements and new features for users, they also modify how applications collect, process, and store user data on the device. In response to each application update, forensic tools must also update to support the new ways applications store data on the filesystem. If no tool is available, the forensic examiner must manually reverse engineer the application to determine how and where it stores data on the filesystem.

1.1 Motivation

Finding faster method to reverse engineer application file formats motivated this research. Knowledge of how and where applications store data on the device is good for forensic examiners to extract digital information during an investigation. This is a large and growing field of research sine applications process, disseminate, and store data in many different ways. For example, the out of band communication,

end-to-end encryption, and unique storage solutions of some chatting applications provide criminals the privacy and anonymity needed to coordinate, conduct, and plan nefarious activities [15] [16] [17] [18].

Finding faster methods to reverse engineer application file formats motivated this research to develop an automated solution to reverse unknown mobile application file formats. The out of band communication, end-to-end encryption, and unique storage solutions that some of the chatting applications provide criminals the privacy and anonymity needed to coordinate, conduct, and plan nefarious activities [15] [16] [17] [18].

Forensic examiners have a challenging time analyzing applications that advertise their purpose for secure chatting. Additionally, there is a whole class of applications that mask their intent behind a euphemistic facade. For example, a 2015 court case [19] involved several high school sexting pictures using an application disguised as a calculator [20].

Furthermore, applications exist that have a chatting feature as a secondary purpose. For example, gaming applications, such as Zynga Words with Friends[21], social dating applications, such as Tinder [22], and navigational applications, such as Waze [23], allow for users to chat online with each other, and thousands more like them.

The reliance on phones for everyday tasks is at an all time high – users rely on their mobile devices for communicating, navigating, banking, and other day in the life tasks[24]. These routine user activities leave behind forensic artifacts on the phone storage that if stitched together correctly could be used to analyze typical user behaviors.

Due to increased use and evidentiary contents, the Department of Defense (DoD) is motivated to invest in more efficient and cost effective solutions to access and extract digital evidence from mobile devices. For example, in response to the terrorist attack

in San Bernardino, California, the FBI issued a court order for Apple to unlock the shooters iPhone and extract forensic data that would be relevant in the criminal investigation [25]. After entering into a political battle with Apple, the FBI went another route and purchased services at $900,000 [26] from Cellebrite [27] to break into the terrorist used device to extract digital evidence. Cellebrite is an Israel-based vendor that specializes in digital mobile forensics [28].

Several other government agencies have invested in digital forensic software and services from Cellebrite. The Federal Bureau of Investigation (FBI) signed 187 contracts valued at $2 Million dollars for mobile forensics products and services from Cellebrite in 2016 [29]. Other agencies that have used their services include the Department of Energy (DoE), Drug Enforcement Agency (DEA), Internal Revenue Service (IRS), military services, and government departments [26].

The Department of Justice (DoJ) relies on digital evidence accessed and extracted from mobile devices to prosecute criminals and defend the innocent in court [30]. In a 2017 case, police use geolocation data from a suspect's phone to convict the murdered wife's husband of homicide [31]. After serving two decades for a murder conviction, key cell phone geolocation acquits the wrongly accused in 2018 [32].

1.2 Methodology

The purpose of this research is to develop a process that automates the analysis of an application to determine how and where it stores evidentiary data in files. The goal is an implemention of the new proces that reduces the extraction time required for an examiner to collect information from an application's files.

The hypothesis is that through an automated static with iterative dynamic approach, one can reconstruct the file formats that the application uses to store its data structures to the filesystem. By establishing a linkage between the data structures

used in the application layer and the bytes written to the the file on the filesystem, an object-to-data legend is constructed that links the application data structures to the physical sections of the file. With this mapping, a forensic tool can extract evidentiary information from a given application's data file. This methodology would eliminate the need to manually reverse engineer an application to determine where user information is located in an application's storage files. The forensic examiner can use the revealed file format to reverse engineer an application and all subsequent developer updates.

Due to the open source nature of the operating system and the relevant Linux research into unknown file formats, this research focuses on Android applications that store their data in unencrypted files using unknown file formats. With additional work, the process this research defines can be used to identify unknown file formats of applications executing on other mobile architectures.

The process developed is called ADSS (Automated Data Structure Slayer); the first automated reverse engineering tool that programmatically integrates dynamic and static taint analysis, tracks tainted data through the application and native layers, and determines unknown file formats for Android applications. ADSS automatically hooks the necessary application methods to develop an object-to-data legend that links the application data structures to areas in the file. With minimal effort from the forensic examiner, ADSS identifies the file format (i.e. the byte-level offsets of evidentiary data in the analyzed file).

ADSS speeds the development of tools to parse and extract data from these otherwise unknown file types. ADSS successfully determines the format of files with forensic evidentiary data used by the Android applications Uber (v4.208.10003) and Discord (v6.6.1). The file formats were verified by extracting data from a second device.

1.3 Summary

This dissertation introduces the Android Operating System (OS). It then details the forensic problem of data reverse engineering research for mobile applications. Next, it summarizes related reverse engineering techniques. It then discusses Automated Data Structure Slayer (ADSS). Finally, it provides results from running ADSS on the Android applications Uber and Discord.

II. Background

This chapter provides context for how an application executes in the Android Operating System. Understanding how the application interacts with virtual and native resources during execution to process, store, and collect user data is essential to implementing a methodology to discover unknown file formats used by Android applications. Monitoring the data from user input and processing on the device are the early steps to determining how it is stored on the filesystem. Then the examiner can reverse the data from the file to extract evidentiary information from the user.

This chapter covers Android internals to show how mobile applications interact with the operating system resources to store, process, and disseminate application data. To determine file access behaviors for a given application, the first part of this chapter provides a foundational overview of how applications interact with system resources. This understanding is necessary for Chapters 4 and 5, which presents details on the ADSS' process of monitoring an application and how it allocates memory to hold user information before writing data to the filesystem. This chapter concludes with a summary of current mobile application reverse engineering techniques for static and dynamic analysis.

2.1 Android Primer

This section discusses Android – its early origins with Unix, the operating system architecture, and the internal designs. Other books can be referenced that provide more detail into Android internals; the purpose of this section is to provide context before discussion on advanced reverse engineering techniques employed by ADSS. The end goal is to show relationships between existing reverse engineering techniques to understand how an application interacts with its resources in the Android OS to

decipher complex file storage behaviors.

History.

Since Android's acquisition by Google in 2005, the open-source and high availability of the embedded operating system has consistently increased in popularity among mobile users worldwide [33], even surpassing the Apple iPhone purchase metrics in 2015 [34]. Google employed a radically different marketing strategy for Android than the other leading device manufactures at the time – specifically by fostering an open-source product (Google Play Services and vendor specific drivers are proprietary [35]) that can be employed across a variety of devices, independent of manufactures and cell phone carriers [33].

The Android OS branched from the Unix OS; in 2010, Android departed from the vanilla Linux Kernel due to the power and resource limitations of embedded devices. In 2012, Android and Linux communities agreed to sync both kernel baselines [36]; as of this writing, there still exists differences between both baselines. Specifically, Android lacks a native windowing system, glibc support, and the standard set of Linux libraries. At the kernel level, Android enhances the baseline with Alarm, Ashmem, Binder, Power Management, Low Memory Killer, Kernel Debugger, and Logger subsystems [37]. The historical Linux underpinnings allow a researcher to utilize some Linux-specific reverse-engineering and analysis techniques on Android.

Architecture.

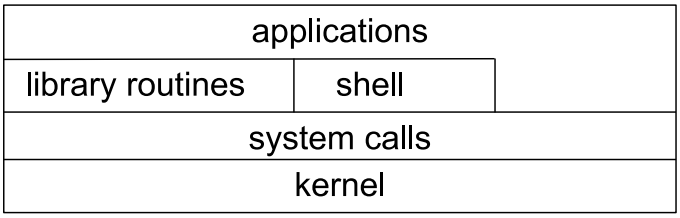

Figure 1. The architecture of the Unix OS [1].

Android maintains a split privilege of execution levels between user and kernel space. Figure 1 shows a high level view of this privilege model. The kernel manages access to the hardware resources for the user-level applications during execution. Kernel level code, meant to be fast and efficient, is smaller in size and contains more permissions than the user-level applications. Applications provide users with the experience they are accustomed to when using an Android phone. The shell is a text-based command language interpreter that executes commands from the user [38].

Internals.

The Android OS environment is organized into several layers of abstraction that operate above the kernel. At the top of Figure 2 sits the Applications, processes that users can launch on their phones.

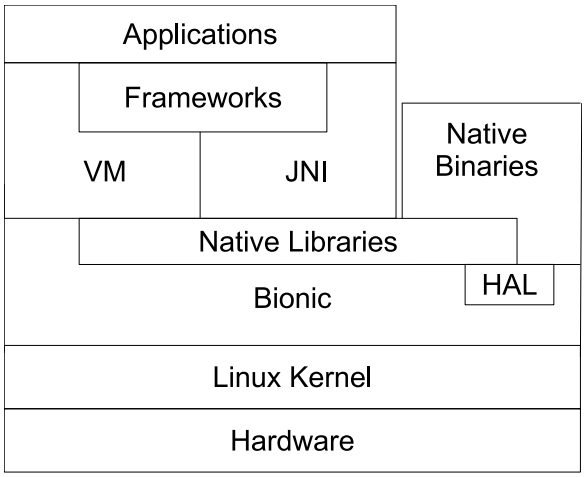

Figure 2. The architecture of Android OS [2].

Developers can write Android applications in either Java-like code, native code (C/C++), or a hybrid of both [39]. Android provides two different environments that support application execution: Android Virtual Machine (AVM) and Native layer . The Virtual Machine (VM) executes files in the Dalvik Executable (DEX) or the Optimized Dalvik EXcutable file (ODEX) format, depending on what version of Android is running. Android 4.0 or higher executes with Android RunTime (ART), while lower versions of Android execute with the Dalvik environment. Inside the VM, applications have access to a repository of code that mirrors the libraries available to Java applications, with the difference lying in the underlying implementation of the runtime environment. Android implemented the VM as register-based, and runs classes compiled by the `dx` tool, a Java language compiler that transforms source files into the .dex format [40] [39], while the Java VM is stack-based. The Native layer allows execution of applications developed in C/C++ and does not require a VM [41]. Applications can access the Native Libraries via the Java Native Interface (JNI) [1]. Having native and virtual machine resources allows developers to design complex applications written in either Java-syntax, C/C++, or both. The Frameworks layer provide programs that manage the basic functions of the phone, such as resources,

voice, activities, locations, etc. [42]. Continuing downward in Figure 2, the Native Libraries layer provide the standard linux open source libraries via the Android VM or the JNI interfaces. Google modified and customized the standard libc library for use in a mobile environment, which is known as Bionic [37]. Next is the Linux Kernel followed by hardware components on the mobile device. Lastly, the Hardware Abstraction Layer (HAL) is a standard interface which allows vendors to standardize device specifications and drivers [37].

2.2 How Applications Work

This section provides an overview of Android applications. It discusses their development, execution, and interaction with the filesystem. This primer is essential to understand in order to grasp the concepts presented in the following chapters. One must first understand how applications are intended to behave before making modifications to uncover how and where they store data to the filesystem.

Figure 3 shows the lifecycle of an application and serves as a blueprint for this section. This section first discusses the major components of an Android Application Package (APK), then shows the application installation process, discusses the Executable and Linker Format (ELF), and then concludes with how the application process executes within its respective virtual machine.

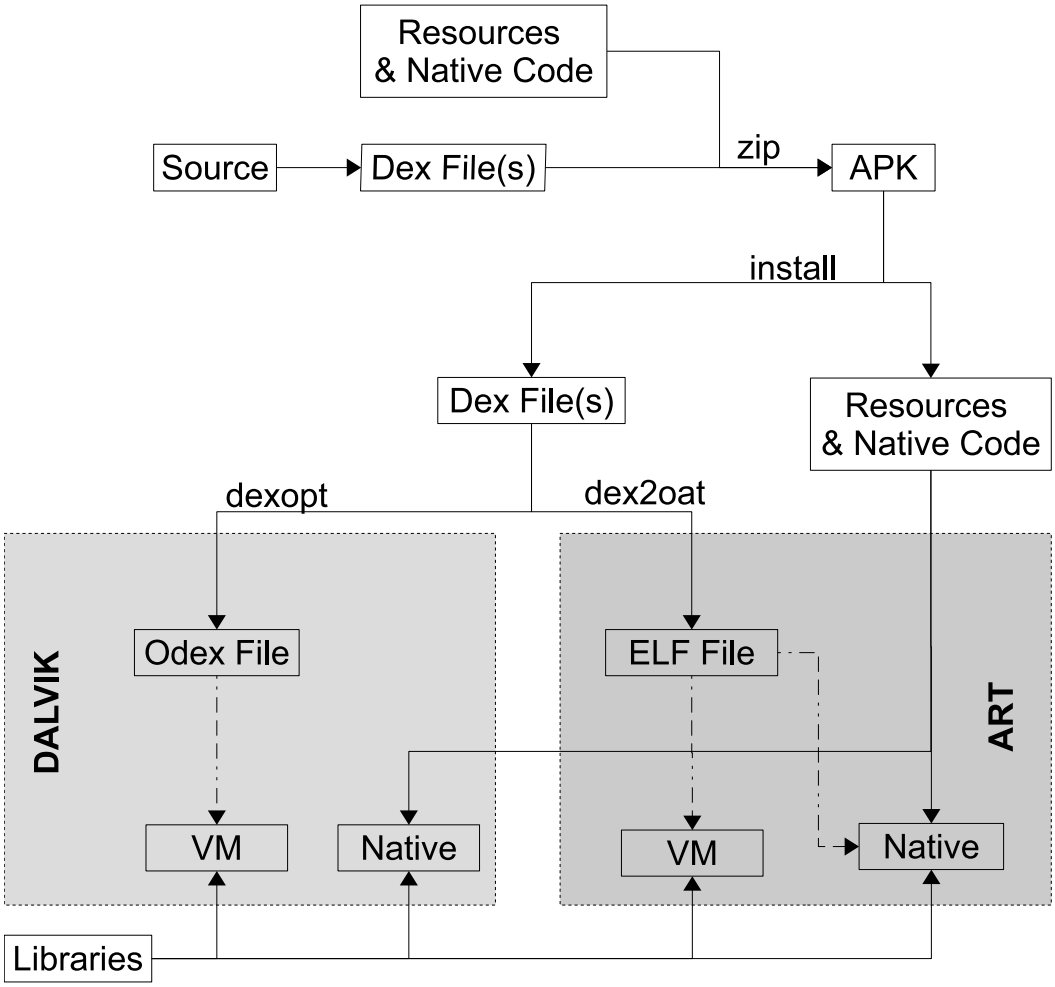

Figure 3. Life of an APK [3].

Architecture.

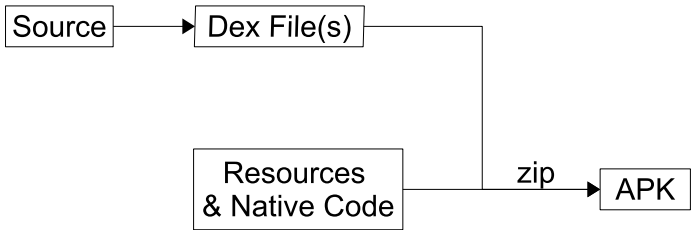

Figure 4. APK creation.

Figure 4 shows the creation stage of the APK lifecycle. The source code of an Android application can be written using three different methods: Android Development

Kit (using Java-syntax), Android Native Development Kit (C/C++), or scripting languages (Perl, Python, Lua, Ruby, Beanshell, JavScript, Rhino,Tcl, Rexx) [43] [37]. Typically, application developers rely on Android or other third party libraries for their applications as this saves development time, promotes code reuse, and improves reliability. The benefit of writing code in C/C++ is mainly performance; this is for gaming developers who want to squeeze every ounce of power and efficiency from the mobile device [37].

Developers can use Android Studio or another platform to build the APK from the source files. The dex files serve as an intermediary step, transparent to the developers, in which the source code is compiled into bytecode to execute within the Android virtual machine. Source code with more than 65K unique function calls and declared class fields require multiple dex files [44]. These dex files, with resources and native code, are compressed into the APK.

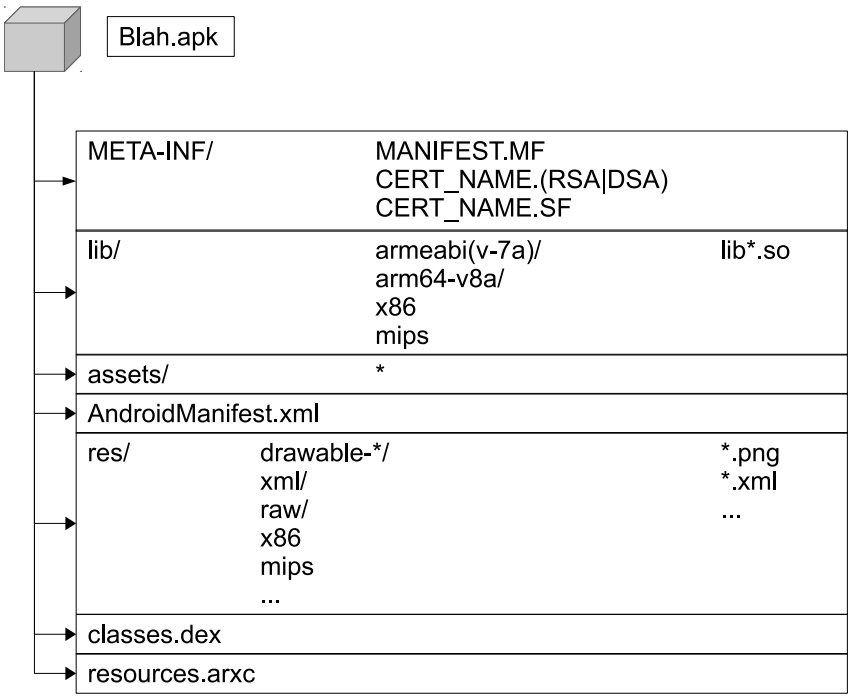

Figure 5. Android APK hierarcy [4].

Figure 5 shows in more detail the components of an APK. Within the 'META-INF' folder, there exists the MANIFEST.MF file, self-signed certificates, and the developers public signature and additional miscellaneous information. The 'lib' folder contains native Executable and Linker Format (ELF) binaries that the application calls upon during execution. The 'res' folder contains resources such as Activity layouts, pictures, music files, etc. in the Extensible Markup Language (XML) format. The 'assets' folder contains raw files that are loaded via the AssetManager during the application lifecycle process. The AndroidManifest.xml file is the entry point for the application. It annotates the application's metadata, required permissions, used Intents, Activities, Receivers, and Services. The 'classes.dex' file contains executable bytecode that is run in the Android virtual machine. The 'resources.arsc' file is the compiled Android resource file. The final types of files in the package are shared object compiled libraries; these are native C/C++ compiled files that execute directly on the processor [4].

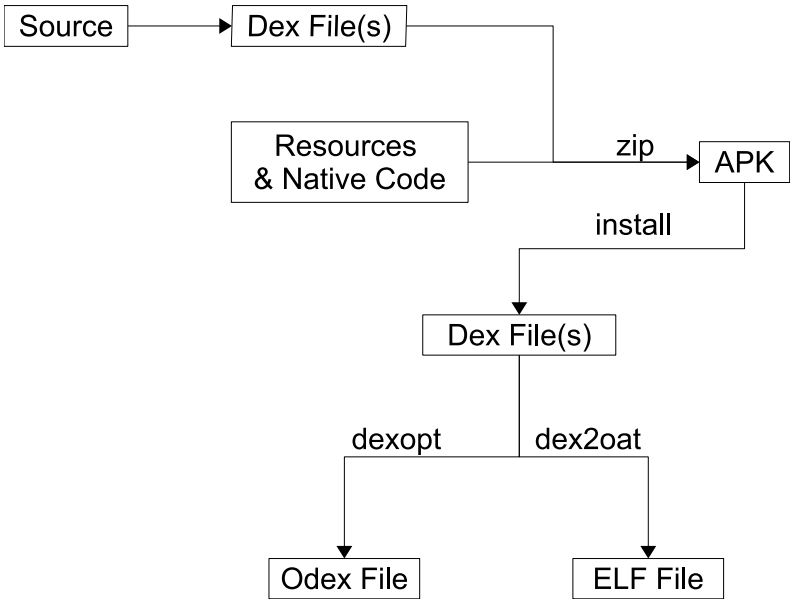

Figure 6. Android APK installation.

Figure 6 diagrams the installation process for the APK. This step decompresses

an APK and the resultant dex files pass through `dexopt` or `dex2oat`, depending on if the Android version supports ART or Dalvic virtual execution environments. ART provides additional optimizations over Dalvik and is compatible with Dalvik's bytecode format. The performance increase of using ART over Dalvik is because `dex2oat` compiles the dex files into native code (ELF executable binaries) before the application executes, whereas Dalvik provides a just-in-time compilation via `dexopt` [3].

This research focuses on Applications executing on Android 4.0 or higher, which uses the ELF fileformat. ELF is an extensible file execution format that provides a common set of standards that hardware vendors must adhere to, but at the same time allow hardware vendors to provide implementations that are specific to their architectural platform.

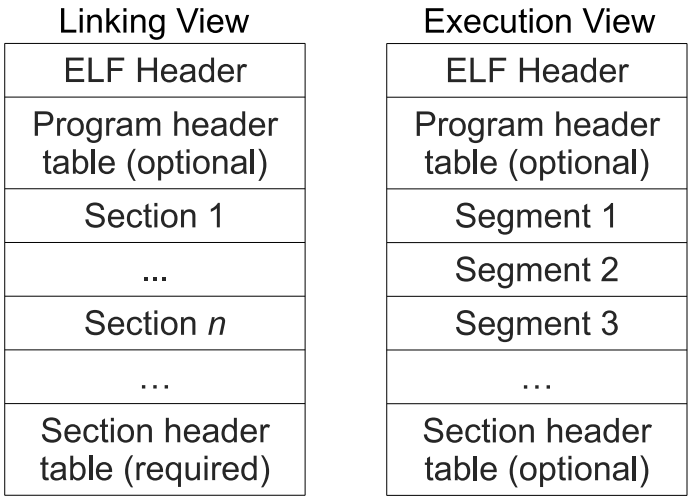

Figure 7. ELF during linking and execution views [5].

As shown in Figure 7, an ELF executable consists of several components. During program linking, the file is divided into different sections with a header table at the bottom or at the top. The header table is a lookup index for accessing a particular section. There are over thirty different types of sections that could be contained in

an ELF file. The most relevant sections of the research include:

- `.bss`: Global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
- `.data`: initialized data that contribute to the program's memory image.
- `.dynamic`: dynamic linking information.
- `.dynstr`: strings needed for dynamic linking, usually the names associated with symbol table entries.
- `.dynsym`: the dynamic linking table
- `.fini`: executable instructions that contribute to the process termination code.
- `.got`: global offset table
- `.init`: process initialization code. This code is run first when the executable starts.
- `.iterp`: name of the program interpreter.
- `.plt`: procedural linkage table
- `.symtab`: static symbol table
- `.text`: executable instructions for the program.

Execution.

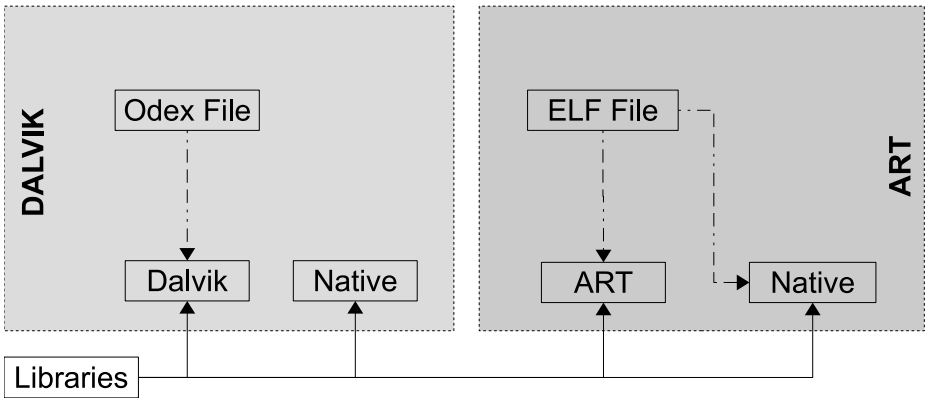

Figure 8. Application executing.

An Android application consists of one or more processes that execute within the operating system. Figure 8 provides an overview showing Android's executable file format (Odex, ELF) running within the respective virtual machine. When a process executes, it accesses hardware resources (memory, Central Processing Unit (CPU), other components) via libraries or directly via system calls into the kernel. An application can be as sophisticated as a game or as simple as an executable that prints "Hello World" to the screen.

This section discusses how processes access resources within Android. The first half of this section discusses native process execution and the second half covers show how processes allocate memory within the android virtual machine. It is important to understand the basics of process execution before applying hooks and monitoring filesystem interaction to reverse engineer how an application stores user data to the filesystem.

Calling `exec()` (system call), Android runs an executable file within the context of the process that started the application. The CPU fetches and executes the native processors' instructions. Typically there are multiple processes executing and vying for CPU time, so the operating system employs a scheduler that manages which

processes get to execute their code on the CPU; sometimes a process needs to wait until a device becomes available (for example reading or writing data to the filesystem) and if another process is ready to execute, then the scheduler saves the context of the waiting process and then load the context of the ready-to-execute process. This way, CPU resources are efficiently employed.

In order to allow for code reuse and provide more flexible options for developers to execute common programming tasks, a corpus of common functions are gathered together and made available to programmers to call; these are commonly referred to as libraries. Android applications bind to libraries either statically or dynamically. During static binding, all library function references are resolved before program execution and the library file links to the binary executable before program execution. The consequence is a larger binary file but also allows for increased application portability – the program executes regardless of if the host operating system has the required libraries. If not statically bound, Android applications can dynamically link to a shared object library at execution. The decision occurs during the application build phase, where the linker neglects the dynamic library's symbols and does not perform checks of the sections (`.bss`, `.text`, or `.data`); instead it checks if the dynamic library contains the symbols needed by the binary. If it finds them, then the linker creates the executable. This is illustrated in Figure 9.

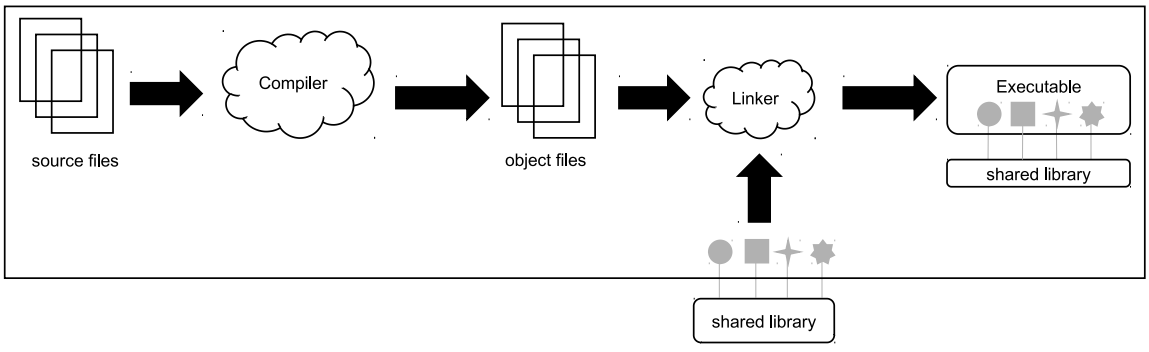

Figure 9. Building with a shared library [6].

If dynamically bound to a shared object, then the Android loader locates the shared library, loads it into the process, and then maps the function addresses from the binary executable to where the library has been mapped to in memory. This is the stage where the shared library segments are integrated into the resultant binary right before execution. The dynamic library loaded at run time must have the same symbols that were generated at build time. The function name signatures (i.e. name, number of arguments, type of arguments, and return values) must be identical to what was generated throughout the build process. Figure 10 illustrates the load time linking process.

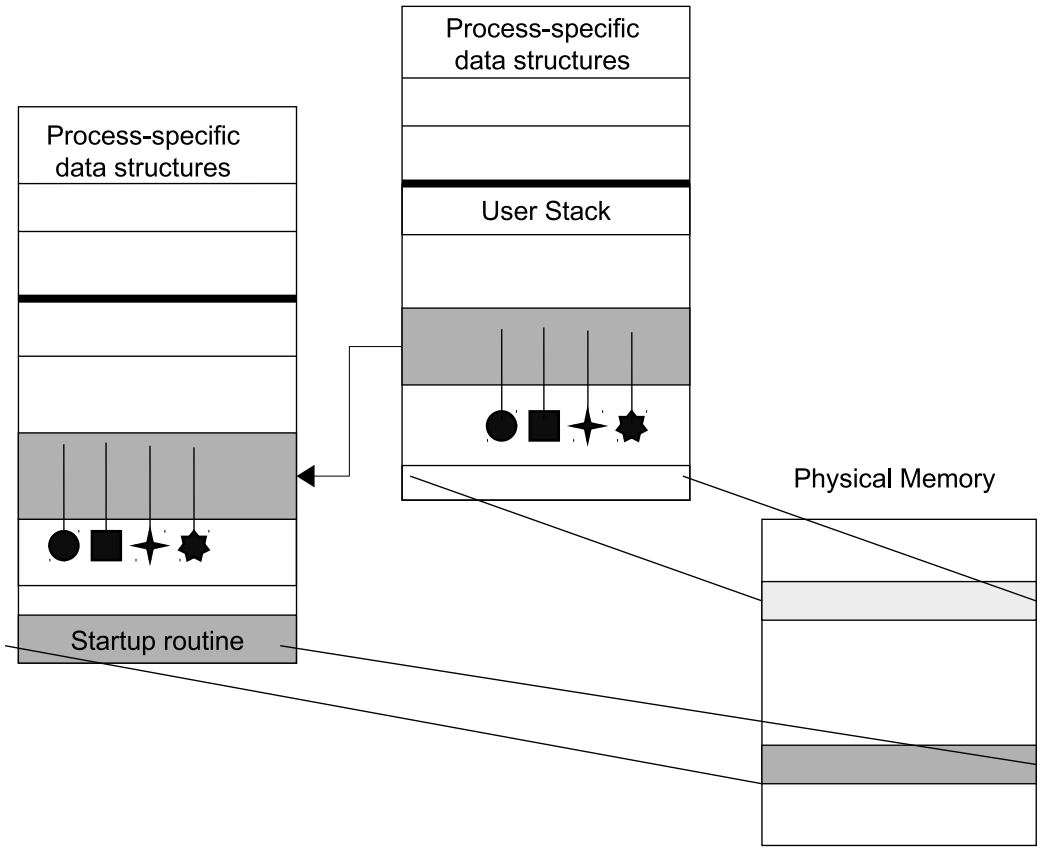

Figure 10. Load time linking of a dynamic library [6].

Different types of memory resources are available to the processor as it executes a processors ' instructions. Registers, due to their proximity to the processor, provide

the fastest means to read and write data, but tend to be limited in storage capacity. System Random Access Memory (SRAM) has slower read and write times, but has larger capacity for storage than registers. The filesytem has the slowest read and write speeds, but provides the largest capacity for long-term storage.

Android carves out a section of memory (SRAM) for the running process [45]. At a higher level address, and typically growing down, is the stack. The stack implicitly stores local variables, temporary information, function parameters, and return addresses that are associated with the running process. The heap allocates dynamic memory for use in the process when certain C/C++ language calls are made, such as `calloc()`, `malloc()`, `realloc`, and `new()`. Deallocation occurs after evoking new and delete. Opposite the stack, the heap grows upwards and holds objects or static variables [7] [6] [1]. Below the heap are the `.text`, `.data`, and `.bss` data that were part of the executed ELF file. Between the heap and the stack is shared memory, where functions from dynamically linked libraries are loaded. In addition to the userspace memory stack, a process has its own kernel space stack. This is an area of memory that program execution jumps to when a system call has been executed or other privileged code is run [45]. Figure 11 illustrates these areas as a native process uses memory during execution [6].

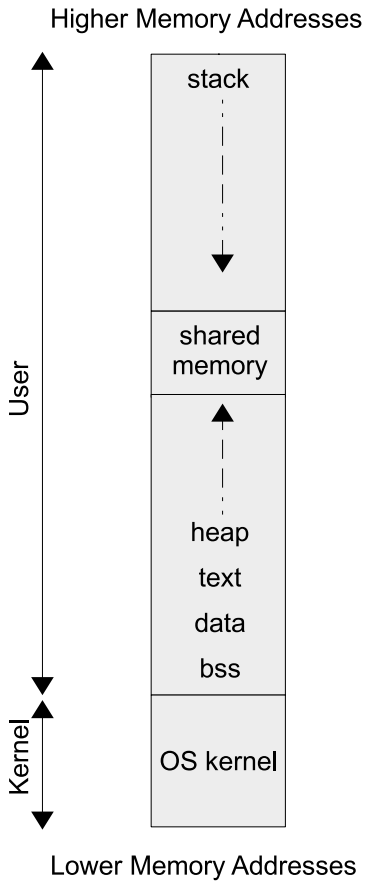

Figure 11. Native application running in memory [7].

The kernel allocates specific data structures during process execution. Every process is identified by the `task_struct` structure, which Figure 12 illustrates. This structure has several fields, but the most notable are `mm_struct`, the outline of the process in memory, `pid_t pid`, which holds the process ID, and `char comm[16]`, which stores the text description of the process.

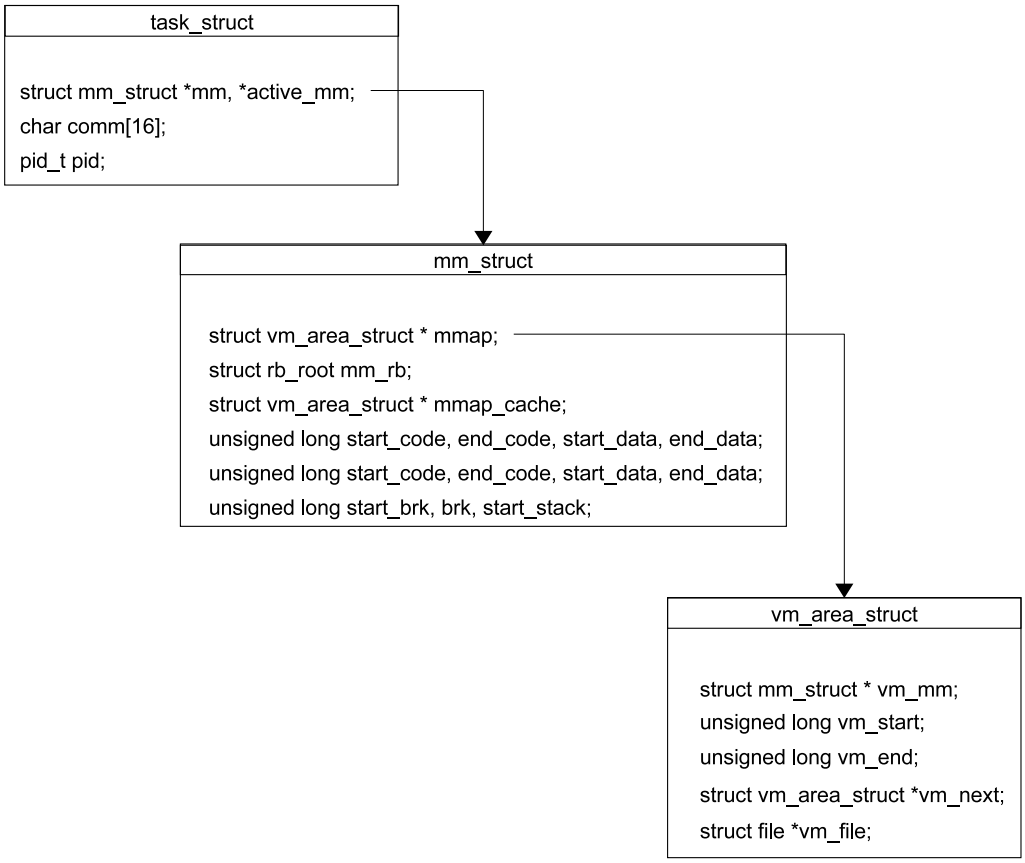

Figure 12. Kernel structures associated with a process [8].

The `mm_struct` is further composed of several virtual memory area structures, shown in Figure 13. Each shared library is mapped to one of these `vm_area_struct`. The heap, stack and the different components of the executing ELF file(`.bss`,, `.data`, `.text`) map to a `vm_area_struct`.

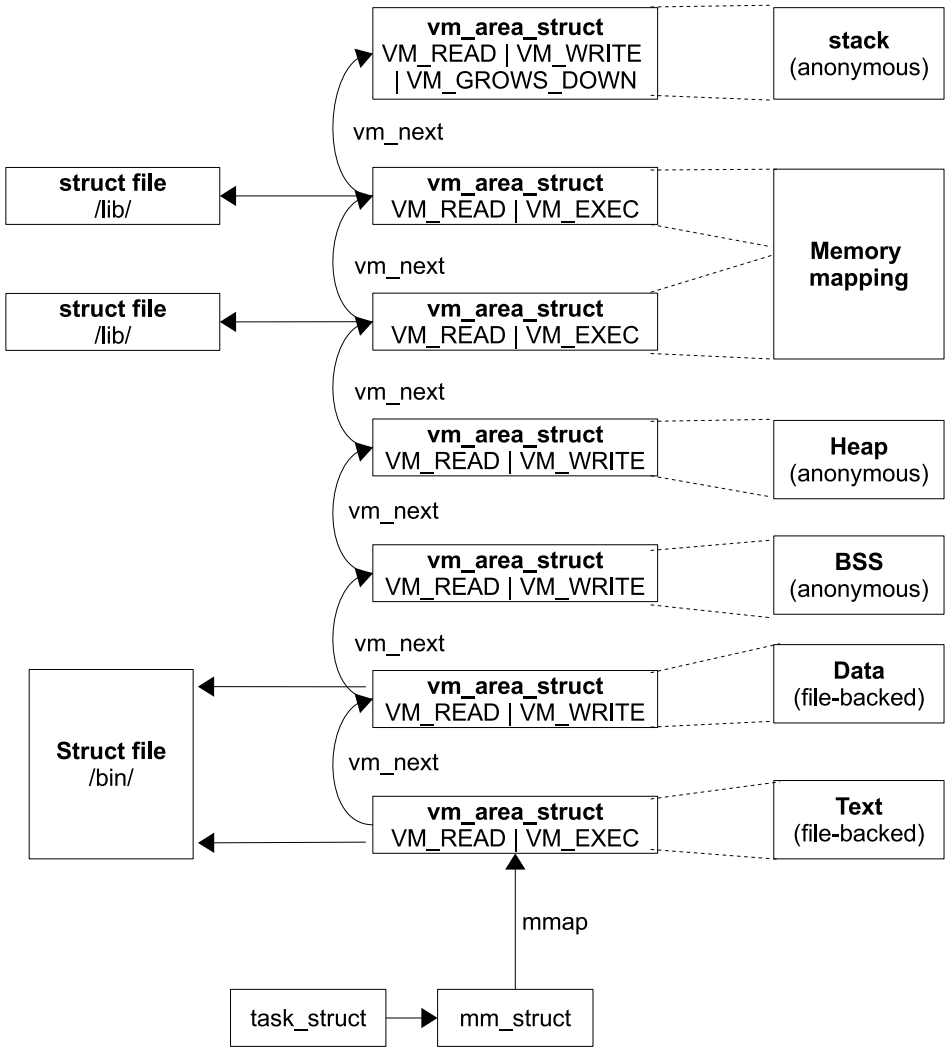

Figure 13. Process segments mapped to memory [8].

As previously mentioned, the `.text` area stores the executable instructions for the program. The `.bss` section stores the global and static variables that are initialized to zero and are not explicitly initialized in source code. Finally, the `.data` area stores any global or static variables which have a pre-defined value and can be modified. The heap, growing upward, contains dynamically allocated memory from C/C++ functions such as `malloc()`, `calloc()`, `new()`, `realloc()`. The .dynamic sections holds dynamic linking information, the .dynstr section stores strings that support dynamic linking (i.e. the strings that represent the names associated

with symbol table entries). The `.dynsym` section stores the dynamic linking symbol table.

In order to provide sufficient resources to a process and to present more memory than may be available physically [9], Android utilizes virtual memory. The operating system shares common resources between processes and utilizes demand paging, which gives processes just in time access to needed resources while storing less frequently accessed resources to the filesystem. As a processor executes instructions it fetchs data stored in memory locations, depending on the particular instruction that it is executing. In a virtual memory environment, the processor has access to a table that provides a relationship between virtual pages and the physical pages in memory. Each of these pages has a unique number and can be referenced via a Page Frame Number (PFN). In the abstract virtual memory diagram, Figure 14 shows two processes and their respective page tables that provide a translation from a virtual page to a physical page in memory. There is additional information in the page table such as a valid flag, access permission details (read, write, etc.), and the physical page entry that the virtual page refers to. If the process accesses a virtual address for which there is no valid translation to a physical page, then a page fault is generated. For example, there is no entry in process A's page table for VPFN 2; if process A attempts to read from an address within this Virtual Page Frame Number (VPFN), a page fault occurs. If the virtual address is valid but the referrrenced page is not in memory, the operating system loads the appropriate page into memory from disk. As a process needs further access to valid data that has not been loaded into virtual memory, continued page faults occur and they are loaded just in time. This iterative process is referred to as Demand Paging [9].

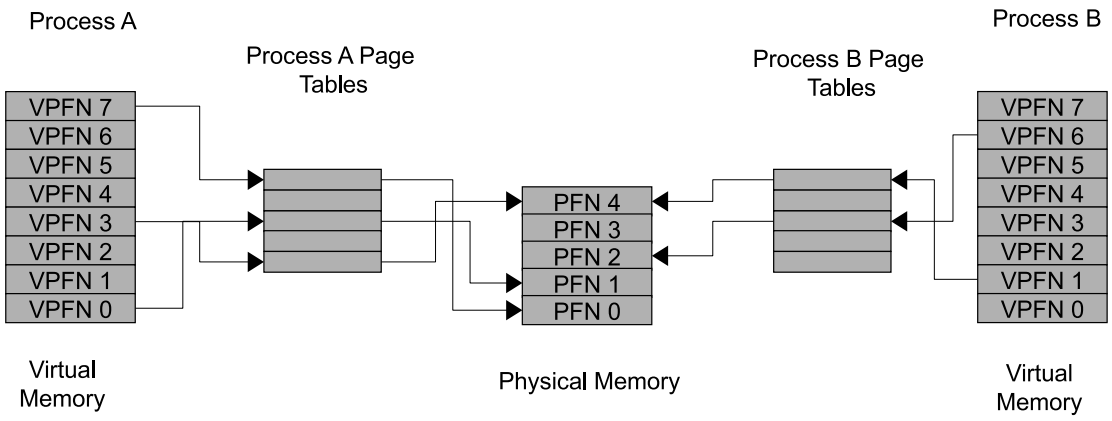

Figure 14. High Level view of Linux virtual memory [9].

Application flow jumps around to various locations in the process code via the use of CPU registers and operating system pointers. The format, syntax of CPU instructions and the number of registers available are architectural specific and defined by the processor Application Binary Interface (ABI). The ARM 64 bit processor, consistent with the AARCH64 specification, provides thirty-one 64-bit general-purpose registers accessible at all times and in all exception levels. These are referred to as registers X0-X30. ARM64 is the processor in the Nexus 6P, the device used for research in this prospectus. Each 64 bit register can also be used as a 32 bit register to support 32 bit applications. The 32-bit W register forms the lower half of the corresponding 64-bit X register. Table 1 shows how Android stores primitive C types in the AARCH64 registers. In addition to the general purpose registers, AARCH64 uses several special purpose registers, as shown in Figure 15 [10].

Table 1. Variable width for registers for AARCH64 used by 64 bit Android OS [10].

type	Size in bits
char	8
short	16
int	32
long	64
long long	64
size_t	64
pointers	64

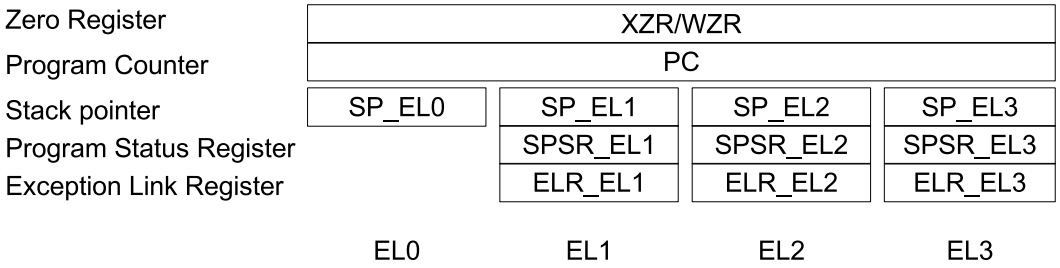

Figure 15. ARM AARCH64 Special Registers [10].

Memory Allocation.

A hybrid application, one which uses resources in the AVM and native libraries, can have complex process heaps and stacks within the virtual environment and the native process environment. Understanding how applications use and process data prior to long-term storage in the filesystem is the first step toward understanding application custom fileformats. These hybrid applications utilize the JNI Bridge to call APIs depending on what mode is executing. This bridge allows for a process executing within the AVM to call native library APIs for for native code to access

APIs defined in the AVM. The `libnativehelper.so` library facilitates these JNI calls [33]. Because the Android Runtime environment is similar to the Java Runtime Environment, understanding how Java allocates memory on the stack and heap is essential to understanding how Android applications allocate memory on the heap and stack.

Pankaj's Java Heap Space vs Stack Memory Allocation in Java [11] shows how objects are allocated in the Java VM heap during a function call. The Java Stack memory is used during the execution of a thread and stores references to objects that have been allocated on the heap. It is when objects on the heap have no more references to them from a process' thread that a Garbage Collector deallocates that object, or rather frees the space in the heap for future allocations [46][47]. Consistent with the native process implementation of a stack, this memory structure is LIFO, which means that when a method call occurs, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

Listing II.1 is a simple class defined in Java that creates a primitive type `i`, two objects of type `Object` and `Memory`, and makes a function call to `foo`.

```java
//Example taken directly from:
// http://www.journaldev.com/4098/java-heap-space-vs-stack-memory
package com.journaldev.test;

public class Memory
{
    public static void main(String[] args)
    {
        int i=1;
        Object obj = new Object();
        Memory mem = new Memory();
        mem.foo(obj);
    }

    private void foo(Object param)
    {
        String str = param.toString();
        System.out.println(str);
    }
}
```

Listing II.1. Memory allocation example.

Figure 16 shows the memory layout of the process in both the heap and the stack. The objects are allocated in the heap and the contents of the primitive types are in the stack. This is conceptually a similar memory layout to how an Android application would store its primitives and objects during program execution.

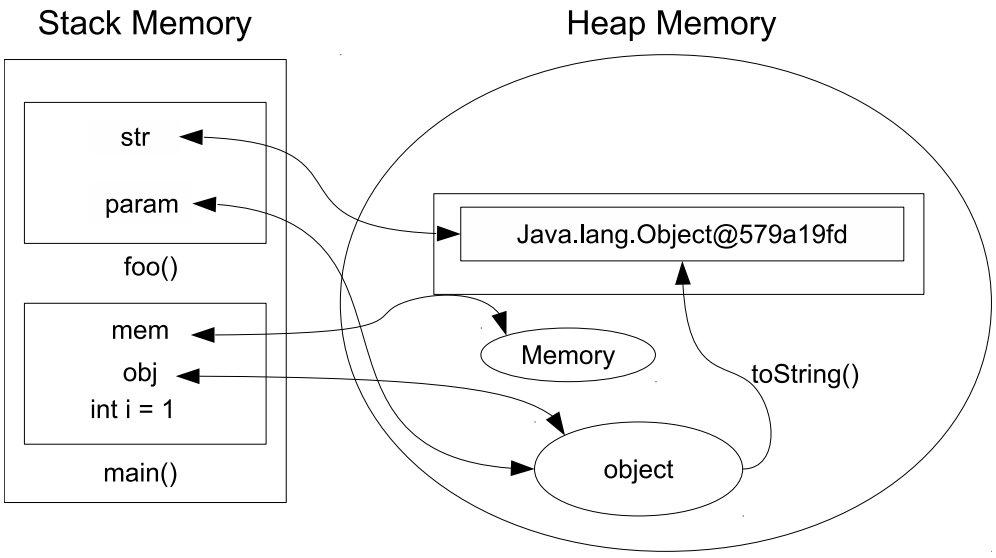

Figure 16. Java Heap and Stack space example [11].

Filesystem Interaction.

This section provides an overview of how applications store data to the filesystem. Android permits applications to store data to either the internal or external storage of a device. External refers to removable media such as a microSD card but for devices that do not support external media, external refers to a logical partition of the primary storage. Android divides the internal filesystem into directories, with each one holding data from a specific application. Android security measures ensure only the application can access the data under its respective folder. Due to this security policy, developers design applications to store the sensitive data under the internal filesystem. On the other hand, Android allows application to store data anywhere in the external filesystem.

There are no restrictions on the structure of the data an application stores to the filesystem. Android allows applications to store private data in SQLite databases; this is a popular method for storing internal data for an application [48]. The structure SQL provides allows for forensic tools to easily parse and view the information using tools such as sqlitebrowser. However, applications can write the data to the tables of the database in any format – which means that table fields can contain encrypted, obfuscated, or raw data in the Binary Large OBject (BLOB) format [49] – this makes data extraction non-trivial. Other third party storage options exist that an application can use to store data such as Kryo [50]. This uses a custom Java object serialization process to store application objects to files using a key-value format[50]. Other non-trival file formats include use of SQL Cipher [51], an open source project that encrypts the database when not in use. Moreover, an application can use system calls or other methods to directly write data to plain files wit any custom file format.

In summary, there are many different ways to store application data with the only restriction being that an application can only store data within its respective internal

storage folder.

2.3 Reverse Engineering

Most developers do not make their source code public and thus analyzing an application's inner workings via reverse engineering is the only way to determine how and where an applications stores data to the filesystem. This section explores the common techniques available for statically and dynamically reverse engineering an application. The first half is devoted to static analysis techniques followed by a discussion of dynamic reversing techniques.

Traditional differences between dynamic and static program analysis still hold true when applied to mobile applications: analysis of a program that is performed with and without program execution, respectively [52]. For the scope of this paper, static reverse engineering refers to analyzing a mobile application without running the application and dynamic reverse engineering refers to analyzing an application's behavior during execution.

Static Techniques.

Static reverse engineering provides the examiner with knowledge of the layout of the application without execution [52]. This includes translating binary machine code into disassembly or abstracting disassembly into a higher level programming language [53]. With regard to Android applications, this section divides static reverse engineering into five steps: Access, Unpacking, Dissimilation, Building, and Signing. The following paragraphs discuss the goals of each step and provide available techniques to accomplish the respective step's objectives.

1. Access

   The first step is to gain access to the application to analyze. www.apkpure.com and similar sites that mirror the Google Play Store provide a way to download application files without requiring a mobile device. If a mobile device is available, an examiner can retrieve the application directly from the phone via the Android Debug Bridge (ADB) [54]. This method requires elevated root level access [55]; it allows the examiner to interact with the phone's filesystem and pull the application from the phone to a computer for further analysis.

2. Unpacking

   After acquiring the APK, the examiner decompresses it to expose the underlying files and first level structure. Android uses the ZLIB [56] compressed package file format for distribution and installation of mobile applications. ApkTool [57], IDAPro [58], and standard operating system decompression tools such as unzip [39] can unpack the APK.

3. Dissimilation

   An examiner may disassemble or decompile the unpacked `.dex` file(s). Both ways break up the `.dex` file(s) into understandable pieces of information, and help the reverse engineer better understand the inner workings of the application. The first method disassembles the `.dex` file(s) into Smali [59] code. Smali is an intermediary-register-based language between source and the decompiler which converts bytecode into Smali. BakSmali converts Smali back into bytecode that is represented in a `.dex` file [4] [59]. The Smali language provides the examiner with low-level access to how the application uses registers, memory locations, strings, methods, classes, etc within the Android Virtual Machine. An Android application can be disassembled into Smali and then reassembled

back into the dex file using BakSmali or ApkTool.

Other tools, such as jadx, can decompile bytecode into a high level language to get the code as close to original source as possible. One can read this source code using a Java Integrated Development Environment (IDE). Viewing high level source is easier to understand the logic and flow of the program than analyzing lower level assembly / Smali for the reverse engineer. Unfortunately, some information is lost during the decompilation process, and thus it is not possible to return from decompiled code to a functioning bytecode in the same way that BakSmali can re-assembled the Smali files[39]. Other tools that can perform decompilation include JEB, IDA Pro, and DAD [39] [4].

4. Building

   During this phase, the examiner reassembles the Smali and other files back into an APK package using ApkTool or BakSmali.

5. Signing

   During this phase, the APK package can be signed using Java jarsigner.

Dynamic Techniques.

Static reversing only provides information about an application, but it is the process of watching how and when the application writes to the filesystem that yields progress toward uncovering the closed-source-file formats. Dynamic reverse engineering allows an examiner to monitor how an application interacts with the operating system resources while it is running [52]. This section provides an overview of several dynamic reverse engineering techniques for Android applications.

One technique, Dynamic Binary Instrumentation (DBI), allows a reverse engineer to inject custom code inside of a running process with a technique called library

injection [60]. Collin [61] relies on a standard Linux debugger, `ptrace`, to monitor the execution of a process and then when a process loads a library the system injects a custom library in its place. This technique allows for the researcher to inject custom code into the process space of the analyzed application. This injected code can be used to dump the stack trace, show the objects that the current thread allocated in memory during program execution. Many things can be done by injecting code into a running process; it is this technique that the future chapters continue to develop.

Another technique is to analyze how an application behaves with other applications. Applications can communicate to other installed applications using four different methods: passing intents, accesing content provided by another application, accessing broadcast receivers/events, and using a service exported by another application [62]. Drozer [63], a security tool that dynamically interacts with an application using a command like interface, deploys a custom application (ie the Agent) that allows the examiner to communicate with the targeted application via the agent. The examiner can send intents [64], access broadcast receivers/events, services, and other content available by the targeted application. This method allows the examiner to analyze the behavior of the application with neighboring applications.

The final method, taint tracking, observes differences in network traffic, data stores, or other activities by inserting a tag into the transferred data that allows monitoring to occur through the appropriate processes, system, or application components. Based on the examiner's objective, tainting can be applied at different levels to achieve different results: operating system, execution language, and the source code [65]. Dynamic taint tracking is most revealing the closer it is applied to the information that the reverser wants to observe. Bell and Kaiser [65] developed a Java-generalized taint tracking engine, Phospor [65], which can be applied to taint an Android's application as it interacts with the underlying Dalvik virtual

Machine. TaintDroid [66] modifies the Android OS source code to taint private user information that may be leaked during an application's usage. Droit [67] performs taint analysis at the binary and Dalvik virtual machine layer of the Android OS by monitoring ARM instructions emulated by QEMU. Similarly to TaintDroid, it heavily modifies the AOSP source code. Finally, ARTDroid [68] is a research endeavor that provides a framework for hooking virtual method calls supported in the ART environment. Although it is not a taint tracking engine exactly, it does provide dynamic execution flow techniques that can be used to monitor how an application behaves in the ART virtual machine without incurring performance penalties associated with instrumenting the application's Java bytecode.

This chapter provided a foundation for the reader to understand how applications are loaded, executed, and interact with the Android OS. Moreover, this chapter discussed the reverse engineering techniques currently available to provide static and dynamic analysis.

III. Related Work

This chapter provides a summary of related work that support identifying the file formats of closed source applications. Related works that completely encompass the scope of this topic exclusively for Android applications are scarce, however there are research papers which support different subset-areas of this research.

The chapter first presents static analysis of binary executables. Following, it summarizes works to show dynamic data flow analysis using taint tracking or method call hooking in a virtual machine. With this understanding, the next section details type tracing in a purely native execution environment. This chapter concludes with research showing the integration of both dynamic and static analysis to monitor code execution and to determine data structures.

3.1 Static Analysis of Binary Executables

Static techniques exist to extract data structures from a binary executable. One, CodeSurfer x86 [69], uses Value Set Analysis [70] to recover intermediate representations that a compiler could generate from an application developed in a higher programming language [69]. CodeSurfer x86 is a system that interfaces with the binary disassmbled code via an IDA Pro plugin called Connector. Connector uses the IDA Pro API to access the call graph, static memory address, calls to library functions, and procedural boundaries. This data is then pushed into a Value Set Analysis (VSA) [71] [70] algorithm. According to [69], 'VSA is a combined numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values or addresses that each abstract location holds at each program point. The set of addresses and numeric values is referred to as a value-set. A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously.'

CodeSurfer takes this data and generates dependency graphs that aid in static analysis: Control Flow Graphs, System Dependence Graphs, and Procedural Dependent Graphs. These dependency graphs help determine instructions that are dependent upon each other but may not be juxtaposed to one another. [69].

File-Format Extractor for x86 (FFEx86) [72] is a static analysis system that builds upon the work from CodeSurfer x86. This project uses VSA and Agregate Structure Identification to annotate Hierarchical Finite State Machine (HFSM) that is used to partially characterize some of the binary executable output data values.

FFEx86 reviewed the source code from common Linux programs (gzip, png2ico, compress95, tar, and cpio) and categorized that these programs write to disk in either a bulk or an individual manner. Individual writes can be conducted using Standard Input Output (IO) functions such as `fputs` or `fputc` or by using wrapper functions (i.e. custom write functions that would write individual bytes of data). During a bulk write, the programmer develops a structure with custom headers, data sections, and additional fields; after construction the entire structure is written to disk at the same time.

The FFEx86 system analyzes the dissasembled code from the IDA Pro Connector plugin (part of CodeSurfer x86) and classifies all output functions (ie wrapper functions or calls to these functions that generate output data) as an FSM. FFEVx86 then creates a redacted interprocedural control-flow graph (or an HFSM) to characterize enterance nodes, exit nodes, call nodes, and all output operations [72]. According to the researchers, the HFSM greatly reduces the complexity of a full call graph of the targeted executable.

FFEVx86 relies heavily upon CodeSurfer x86 ASI and VSA in order to associate structures with FSM data outputs and determine output data structures. FFEx86 does this by over approximating the output file format from the program's specifi-

cation, rather than producing an exact copy of the output data structures – ie the language of the outcome is a superset of the output language of the executable being analyzed [72].

FFEVx86 was successfully evaluated against three open source Linux programs (gzip, png2ico, and ping) and even determined a discrepancy between the documented output for png2ico and the actual output file structure [72]. Limitations of FFEX86 include limited architecture support (ie only for x86), testing against a small set of programs, and only evaluating output operations without considering input operations [72].

These static analysis techniques are limited to the x86 architecture. Also, they are unable to monitor how executed code interacts with the operating system via system calls or dynamic function calls. Furthermore, static analysis is unable to characterize the data structures stored in memory during execution (dynamic, conditional, or simple structures).

3.2 Virtual Machine Data Flow Analysis

Based on the research from Bell and Kaiser [65] and their development of Phosphor, a general purpose taint tracking engine that can be deployed against a variety of Java Virtual Machines, they categorize taint tracking engines into monitoring environments that taint data at different levels: operating system, execution language, and the source code. Bell and Kaiser observed that traditional techniques for dynamic taint tracking reveal more relevant information the closer to the source code the data is tainted. The Phosphor taint tracking engine was designed to be portable across multiple architectures that implemented the Java Virtual Machine.

Phosphor relies upon ASM [73] to instrument the application's Java bytecode, keeping track of the variables accessed by storing a tag for every variable and each

derived variable during the execution of the program. This research is applicable toward enhanced debugging, end-user privacy testing, and data security. [66] Phosphor is efficient and portable and requires no modifications to the Java Virtual Machine (except in the case of Android's Dalvik in which minimal modification occurs); it has been tested against a variety of Java Virtual Machine instantiations, such as IcedTea, openjdk, and Android's Dalvik virtual machine. Phosphor limitations include only tracking dynamic flow and not control flow; implicit data operations are not tracked, only explicit operations. In addition, Phosphor is limited to the confines of the Java Virtual Machine and does not taint variables during native execution. [66]

TaintDroid is a long standing open source project designed to identify private user information that may be leaked during an application's usage. Specifically, TaintDroid's project objectives are to monitor location, camera, and microphone user data. Differing from Phosphor, Taintdroid makes many modifications to the Android Operating System source code that explicitly taints application data as it executes and uses operating system resources. This extensive amount of work [66] (over 32,000 new lines of code; 18,000 are in assembly and 10,661 are in C) lends itself to be a big project that must be routinely updated to maintain relevance as Alphabet releases new Android operating systems and patches. Similar to Phosphor, Taintdroid instruments the application's Dalvik bytecode to monitor variables during program execution. However, TaintDroid also monitors interprocess communcations between applications by hooking the Binder Class, which is specific to Android. Although TaintDroid only performs variable tainting at the bytecode level, it does perform method-level tainting, which means that it monitors the input and return type for native library functions. Moreover, TaintDroid taints the information being read and written to the SMS and Addressbook databases. Lastly, TaintDroid identifies when tainted data exits on the network interface by monitoring the Java framework libraries

at the point the network socket is invoked [66].

Droit [67] performs taint analysis at the binary and Dalvik virtual machine layer of the Android OS by monitoring ARM instructions emulated by QEMU. QEMU [74] is an open source processor emulator that is utilized as the emulator for the Android Operating System. Droit is able to dynamically switch monitoring between java objects and native code instructions. Droit bases its monitoring of J ava bytecode on how TaintDroid works by heavily modifying the Android OS source code. It complements this dynamic analysis by also analyzing the QEMU simulated ARM instructions that execute when native code runs. This is a heavy weight solution, but does allow Droit to analyze the behavior of sophistical malware that attempts to hide its actions by running native code.

ARTDroid [68] is a research endeavor that provides a framework for hooking virtual method calls supported in the Android RunTime environment. Although it is not a taint tracking engine exactly, it does provide dynamic execution flow techniques that can be used to monitor how an application behaves in the ART virtual machine without incurring performance penalties associated with instrumenting the application's java bytecode. The concept behind ARTDroid is to utilize the virtual method table (vtable) to modify virtual method execution, in essence hooking the virtual method calls in the ART Runtime Environment. ARTDroid allows one to dynamically discover what java framework APIs are being called by an application and dynamically hook each call to execute your own code.

3.3 Native Data Flow Analysis

Moving out of the virtual machine/interpreter and into native code dynamic analysis, Tupni [75] automatically reverse engineers an input format. Although it is not specific to Android, it does provide necessary intelligent analysis of data flow at the

native execution level, something that ARTDroid, TaintDroid, and Phosphor fail to provide and analyze. The researchers analyzed 10 different media file formats and determined that they store data in arbitrary sequences of chunks [75]. The tool relies on iDNA [76] to dynamically trace the code execution and a taint tracking engine to associate data structures with addresses in the application's address space and update them as the application executes. With this information, the tool tracks fields, record sequences, and sequence of records used in file storage. This approach allows for monitoring at byte level addressing.

The basic algorithm of Tupni first identifies fields, then identifies record sequences, and finally identifies record types [75]. This approach is consistent with Lim's paper Extracting Output Formats from Executables [72] in which they expect a formatted file header followed by a chunk or chunks of data. Tupni identifies all field chunks in the execution trace by stepping through each instruction and monitoring the bytes as they are read by the application [75]. The tool assigns a weight to each chunk that represents the total number of time that an instruction has accessed that particular byte or bytes. This process is specific to x86 architecture and allows it to identify 8, 16, 32, 64 bit integer operands and floating point chunks [75].

Tupni [75] identifies record sequences by assuming that all applications have to use recursive calls or looping to process sequences of records. Tupni specifically looks for the case of loops, and does this by finding cycles in the control flow graph of the application. Tupni then maps the loop information to the execution trace by mapping the program counter to the instructions identified in the program cycle. [75]

Tupni tries to identify the type of record by comparing set of instructions executed during one record against another; if the instructions are the same or if they contain sets of instructions (ie child records) that are identical to the parent or another analyzed record, then those two records are of the same type. In this way Tupni is

able to group together same record types and more accurately identify storage type formats the more files it analyzes. [75]

HOWARD [77], is a solution developed to reverse engineer data structures from C binaries without any need for symbol tables. According to Slowinska [77], there is little research in this area and HOWARD supports both the forensics and reverse engineering communities. HOWARD dynamically analyzes the program; during runtime deduces the data structure based upon how the program accesses memory. It can accurately determine frequently accessed arrays, control loops, variables / functions that make system calls, pointer activity, dynamically allocated memory, and even determine some code semantics based on analyzing well-know function and system calls. HOWARD has several limitations and is not 100% accurate with determining data structures. For example, it does not defeat anti-reverse engineering measures (ie code obfuscation, code that checks for the presence of a VM before running), requires an array to be accessed at least four times before correct identification, and only works for programs running in Linux on an x86 processor architecture [77].

HOWARD builds upon other research such as Reverse Engineering Work for Automated Revelation of Data Structures (REWARDS) that also attempts to reveal data structures used in a binary executable during program runtime. REWARDS uses a variety of sophisticated techniques and REWARDS only works for programs running in Linux on an x86 processor architecture and relies on intel pin architecture to access the targeted program's code as it is run on the processor. REWARDS refers to arguments that are passed to functions of known signatures (such as System Calls, well-documented APIs, and other publicly accessible APIs) as type sinks. REWARDS resolves a type of a previously accessed variable once it is resolved from a type sink. It resolves multiple typed variable instances to the same static abstraction (i.e, all nodes in a linked list share the same type rather than having distinct types). Once

REWARDS has determined the program's data structures, it generates an organization chart that shows a hierarchy of the data structures of how they are laid out in memory.

Whenever the program makes a know system call with known arguments, it is able to map the memory locations. It is able to provide a hierarchical view of memory and an ordered timeframe for when those locations were accessed by applying timestamps during analysis. REWARDS also tracks common library calls and can determine the type of an argument with the prior information of the function's signature of the library function. REWARDS relies heavily on gathering type information from variables during the analysis and is most useful for analyzing binary files that have been compiled in strong-typed languages. [78].

Another research effort that HOWARD built upon, Laika [79], attempts to determine data structures from a program via dynamic analysis. Laika scans a memory dump to look for all potential pointers. It estimates the start location of objects as those addresses that are referenced other places and also over estimates the ending location of the objects. Laika translates the objects from raw bytes to sequences of block types. It then clusters objects with similar sequences of block types. If Laika is successful, it is only able to detect aggregate data structures and not any of the fields or embedded objects in the structure. [79]

In Dynamic Inference of Abstract Types [80], the paper presents a method that monitors interaction of variables and classifies them by an abstract type based upon which one interacts with another. The author implemented two tools for performing dynamic inference of abstract types: DynCompB for C/C++ binary compiled executables and DynCompJ for JVM-files derived from high level languages such as Java. DynCompB uses Valgrind to insert instrumented code into the binary program at runtime. The tool monitors variable interaction at the entrances and exits of pro-

cedures. The tool furthermore relies on debugging information within the executable to locate variables and read their values from memory.

3.4 Integrating Native and Virtual Machine Dataflow Analysis Techniques

There is precedence in literature that discusses the integration of static and dynamic analysis to monitor data flow during program execution. Ruoyu proposes Static and Dynamic Combined Framework (SDCF), in Static Program Analysis Assisted Dynamic Taint Tracking for Software Vulnerability discovery[81], to discover CVE-2007-6454, a string copy vulnerability, in an open source streaming media multicast tool, PeerCast. Ruoyu contests that the use of both static analysis and dynamic taint anlaysis is necessary to uncover all software vulnerabilities; dynamically tainting all traffic through the application would be too high overhead without the assistance of pre-static code analysis. SDCF uses static analysis to identify the areas in the source code where vulnerabilities may be present and then taints the data through these locations to verify the vulnerability presence. SDCF accurately detected all attacks against BufferAttacker, TxtEdit, IrfanView 4.25, Foxit Reader 3.0 build 1120.

To demonstrate more efficiently detecting cross site scripting and SQL injection vulnerabilities in web applications based on the Java Servlet Specification, Zhao [82] first statically analyses the code before fully detecting vulnerabilities using dynamic tainting. Zhao postulates that because SQL injection and cross-site scripting are fundamentally taint vulnerabilities, detecting them requires at least a dynamic data flow tracking approach. By first statically analyzing the code to identify potential injection points, he focuses taint tracking at these locations to detect positive SQL or cross-site scripting attacks. Executing the proof of concept on experiment test data from CWE89_SQL_Injection part of Java (v1.2), resulted in high accuracy and low non-response rates against the Juliette Test Suite and Bookstore applets. Similar to

Ruoyu, Zhao demonstrated the benefits of integrating static and dynamic analysis for code analysis.

Kirchmayr demonstrates the benefits of static and dynamic code analysis to better understand legacy programs which may not be supported or where the source code is no longer available [83].

3.5 Summary

This chapter covered different research efforts to dynamically and statically reverse engineering computer programs. Unfortunately their applications toward our research goal is limited. Most of these efforts focus on natively compiled x86 programs. There currently is no found research to automate the reversing of android applications to determine the file formats used to stored persistent data. Since Android is a complex operating system that allows applications to have access to both native and virtual resources, providing an automated solution to discover how and where data is processed and stored in unknown file formats requires a complex solution.

In the next chapter, the culmination of this research is presented in a system called Automated Data Structure Slayer.

IV. System Design and Implementation

Automated Data Structure Slayer (ADSS) implements several of the static and dynamic reverse engineering techniques discussed earlier in this dissertation to monitor how an application stores data to files of unknown file formats. The intent is to rapidly discover the format of a file used by an Android Application so that the examiner can extract forensic data from a file with minimal manual intervention. ADSS is the first automated reverse engineering tool that programmatically integrates dynamic and static taint analysis, tracks tainted data through the application and native layers, and determines unknown file formats for Android applications.

This chapter presents the two main phases of ADSS: Static and Dynamic and the action the examiner takes to during application analysis. Figure 17 shows the inputs and outputs of each phase. The material in this chapter also appears in *Automated Extraction of Unknown Android Application File Formats*, submitted to IEEE Transactions on Dependable and Secure Computing.

4.1 ADSS Overview

During the Static Phase, the examiner identifies an application on a device and a file that may have evidentiary interest, referred to as the targeted file. The examiner downloads the APK from the device and starts ADSS with the APK and the filename of interest. ADSS decompresses the APK into static files, parsing each programmatic element into a database of language components. In addition, ADSS parses the Android SDK sources, storing the inheritance class structure into another database. Finally, the Static Phase patches the original APK file, ensuring the full filename is output during the Dynamic Phase. The Static Phase sends the patched APK file and the reference databases to the Dynamic Phase.

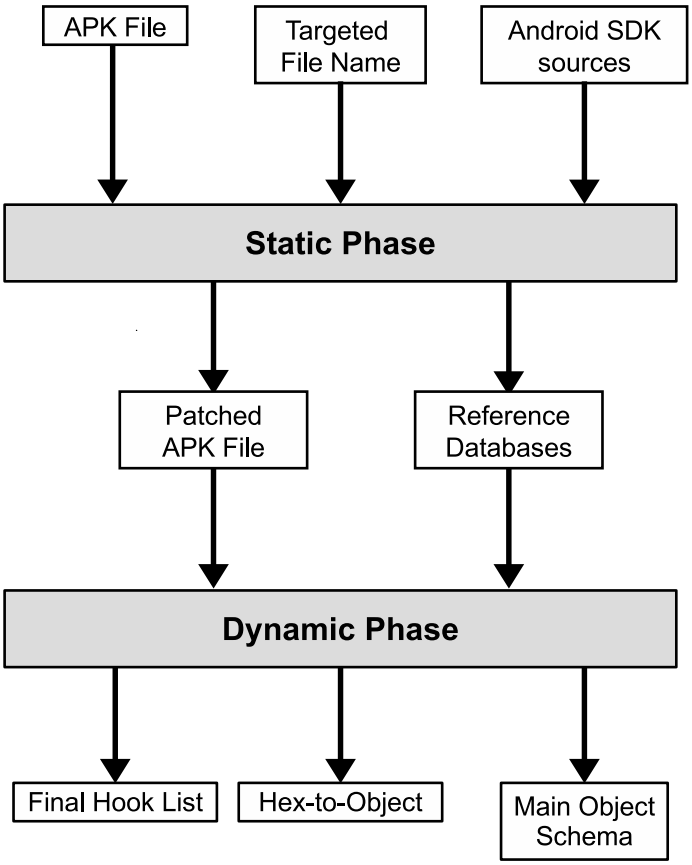

Figure 17. Automated Data Structure Slayer Phases.

During the Dynamic Phase, ADSS installs and launches the patched APK, where it iteratively hooks programmatically discovered key application methods to identify the data structures associated with the targeted file. During the iteration, the examiner will foreground and background the application until the file objects are fully identified. ADSS then outputs the Final Hook List, a list application methods hooked, the Main Object Schema, a tree structure of application objects identified during analysis, and the the Hex-to-Object list, which is the file format of the file that holds evidentiary interest.

This section discusses the implementation details for each ADSS component of the static and dynamic phases.

4.2 Static Phase

Figure 18 provides an overview of the Static Phase steps. The major process steps in the system are numbered in the diagram, with the component name in parenthesis.

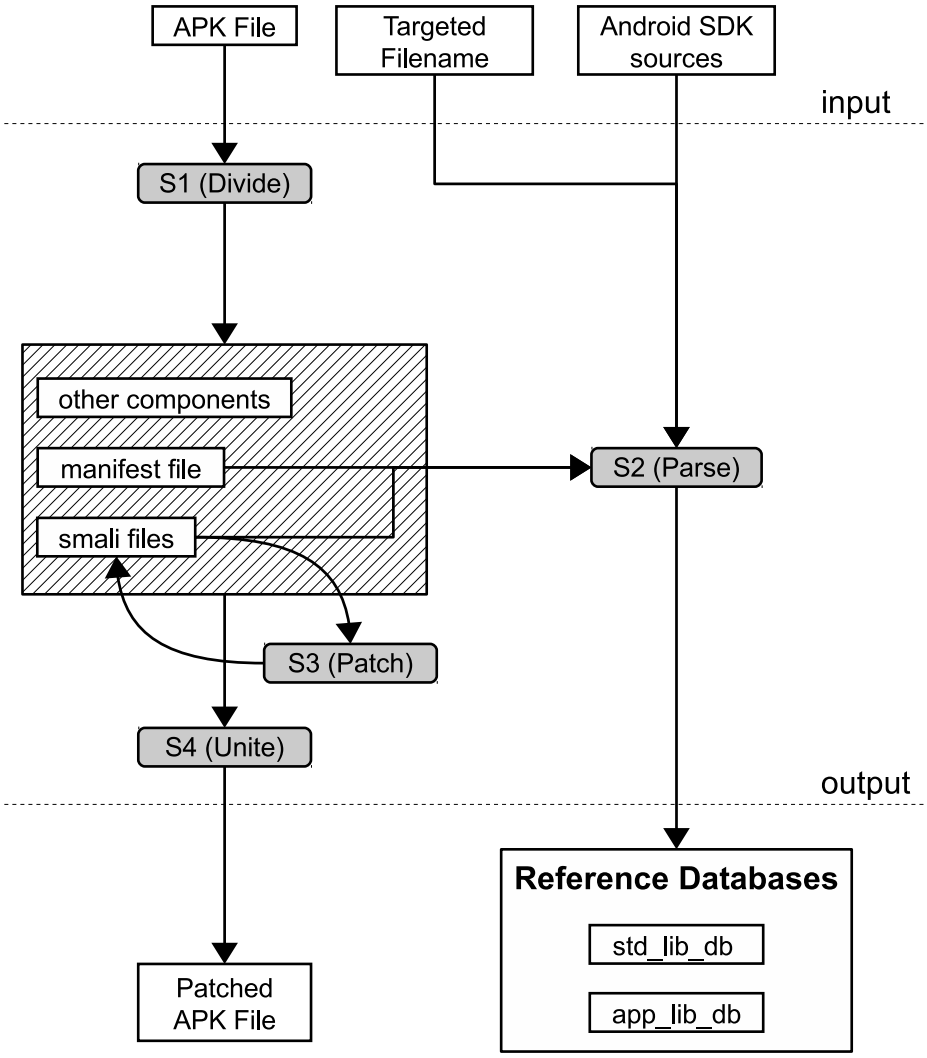

Figure 18. ADSS Static Phase.

S1 (Divide): ADSS decompresses and disassembles the application using ApkTool [57], into its various components. ApkTool is an open source program that unzips the APK into a folder that contains smali code, the original code, the AndroidManifest file, and other component files. It also decodes the AndroidManifest.xml file into human-readable Extensible Markup Language (XML) format. This is consistent with

the static reverse engineering steps Unpacking and Dissimination.

S2 (Parse): ADSS parses the `.smali` files and the AndroidManifest file to extract program language details such as class names, method names, class hierarchy, method calls, fields, permissions, activity names, and other details using a modified parsing engine adopted from smalisca [84]. Smali stores objects and types in registers, where parameters take the form of `p#` and the local variables `v#`, where `#` refers to a register count. The number of registers available is dependent on the opcode, which can have access to a register count of 4, 8, or 16 bits [85]. The registers are available for the scope of the method in which they are defined. Data exists outside the scope of a method in the fields of a class. Table 2 lists the smali primitive types; because Long and Double are 64 bits, they require two registers.

The object types in smali take the form of `LpackagenameObjectName;`, where `package/name/` is the package that the object is in and `ObjectName` is the name of the object. This smali example would be equivalent to `package.name.ObjectName` in the Java language [12]. Since ADSS parses all the smali classes defined in the targeted application, it is able to distinguish between objects from the Android standard library and an application specific type.

ADSS also parses the Android Software Development Kit (SDK) source files to determine the inheritance structure for Android's standard library. ADSS leverages a database of 330 SDK methods (`std_lib_db`), that have been classified such that source and destination are known from all the arguments. For example, `java.lang.System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` takes four arguments, two `java.lang.Object` and three `int` variables. It is unknown without first reading the documentation which array argument is the source and which is the destination. In this particular case, ADSS knows beforehand that the first array is copied into the second array. ADSS uses this

guidance when tracing and tainting objects that pass through standard library calls in ADSS D4 (Trace) and D5 (Taint).

ADSS stores the statically parsed tokens from the `.smali` files and the Android SDK data flows into pickle database files that will be made available later during dynamic analysis. SDK data persists in `std_lib_db` and the application smali tokens in `app_lib_db`. This also ensures that the smali files do not need to be re-parsed.

S3 (Patch): ADSS patches the original application by adding debug information (i.e. the source filename and path into the smali file headers) to the static smali code from each file. ADSS also edits the debug `.line` numbers to ensure they are neither duplicated nor negative in each smali file.

S4 (Unite): ADSS builds, using ApkTool, signs, using JarSigner, and installs, using ADB, the modified smali files onto the device, consistent with the static reverse engineering step: Reconstruction.

4.3 Dynamic Phase

The Dynamic Phase uses the stored data from the Static Phase to programmatically determine the application methods to hook to display the fileformat for the examiner. Figure 19 provides an overview of the dynamic phase steps that make up ADSS. The major process steps in the system are numbered in the diagram, with the

Table 2. Primitive Types[12].

Smali Type	Resolved Type	Size	# Registers
V	void	32 bits	1
Z	boolean	32 bits	1
B	short	32 bits	1
S	char	32 bits	1
C	int	32 bits	1
J	long	64 bits	2
F	float	32 bits	1
D	double	64 bits	2

component name in parenthesis.

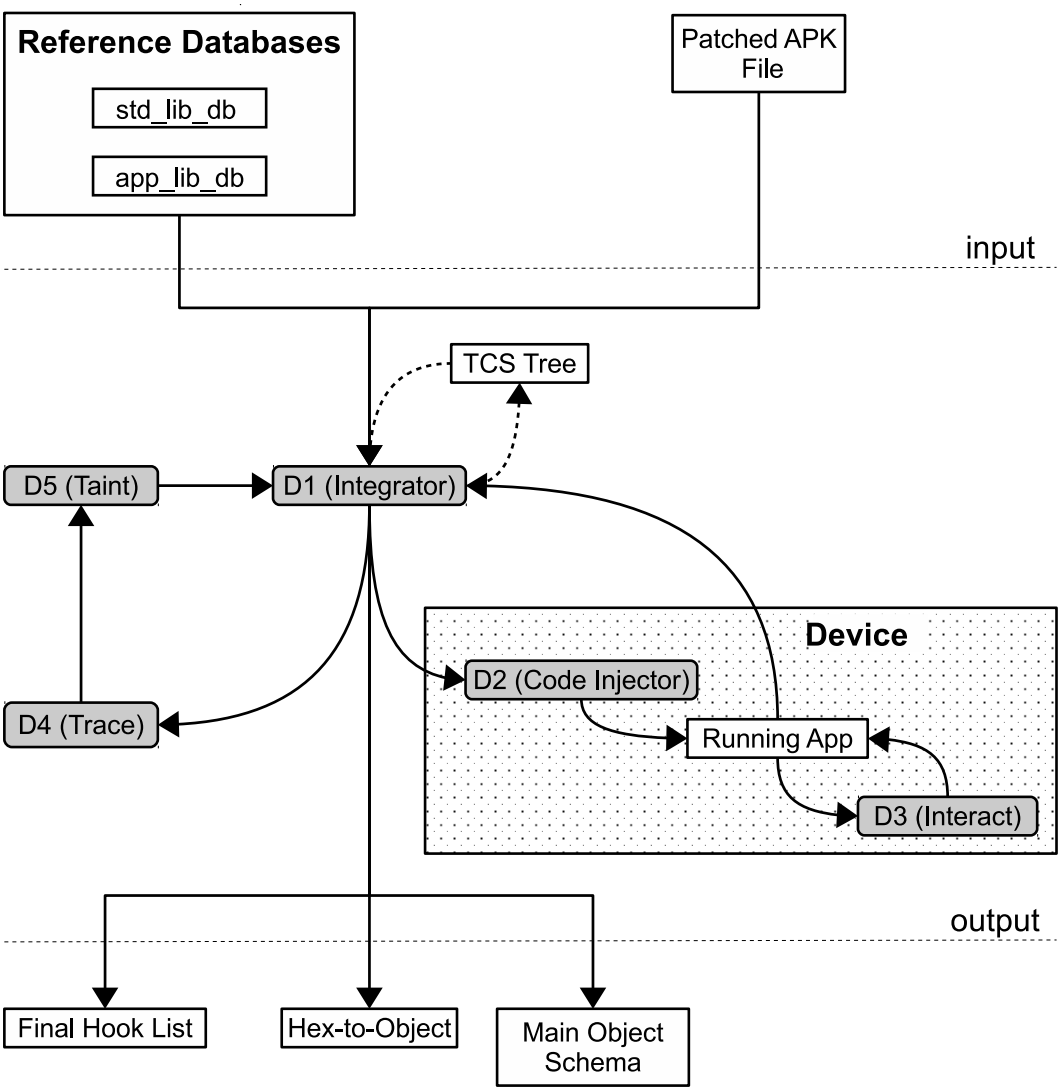

Figure 19. ADSS Dynamic Phase.

D1 (Integrator): The Integrator is the central component in the Dynamic Phase. It uses the tokenized smali code stored in the database during ADSS S2 and directly communicates with the Code Injector (ADSS D2). It injects a dynamic hook into the running program into either the native or virtual execution layers. The Integrator communicates with the Code Injector using a text-based encapsulated protocol, such that each outer layer of meta data is stripped away until the final command is present. This protocol relies on Frida's messaging service [86].

The examiner starts the application and interacts with it until the file of interest is generated in the private storage directory (i.e. `/data/data/application_name/files/`). The examiner downloads the files via ADB and then runs the Linux `file` [87] program on each to identify a file of interest (ie one with an unknwon fileformat such that `file` returns `data`).

ADSS launches the application using Android's built in `monkey script` [88] and pulls back the targeted file via ADB from the application's filesystem. The assumption is the application wrote forensic data to the file and that the file stores the data in an unknown format. ADSS then copies the first 20 bytes of the targeted file to compare during the remaining ADSS steps.

D2 (Code Injector): The Code Injector executes within the Frida server on the phone and directly interacts with the running application. It waits to receive JavaScript code from the Integrator to inject into the target application. Listing IV.1 shows the part of the Code Injector, `onStanza`, that can hook a native or virtual method, as the Integrator directs.

```
1  function onStanza(stanza)
2  {
3    if (stanza.to === '/sampler')
4    {
5      if (stanza.name === '+hook_request')
6      {
7        if (stanza.payload === 'native')
8          run_native_code(stanza.filename);
9
10       else if (stanza.payload === 'java')
11         run_java_code(stanza.code);
12     }
13   }
14   recv(onStanza);
15 }
```

Listing IV.1. Code Injector function `onStanza`.

With a call to `run_java_code`, shown in Listing IV.2, the Code Injector hooks virtual application methods and the code relies on JavaScript `eval` function to run until completions. Hooking virtual functions is further discussed under D4 (Trace) and D5 (Taint).

```
1  function run_java_code(code)
2  {
3    if (Java.available)
4    {
5      Java.perform(function ()
6      {
7        eval(code)
8      });
9    }
10 }
```

Listing IV.2. Code Injector function `run_java_code`.

With a call to `run_native`, shown in Listing IV.3, the Code Injector hooks Android's standard C library (i.e. `libc.so open()`) function, specifically filtering on the filename with the unknown format. Once the targeted filename passes into `libc.so open()` and the associated file descriptor writes to the `libc.so write()` hook, the injected JavaScript code makes successive calls to `java.lang.Thread.currentThread()` and `java.lang.Thread.getStackTrace()`, shown in Listing IV.4. This returns the current application thread's stack trace.

Because ADSS S3 patched the application smali code, it shows the full filename, method name and line number of every call from the standard C library `libc.so write()` to the highest application method that the running thread executed – this is referred to as the TCS Output. It is an unbroken chain of thread function calls that bridges the native and virtual machine layers of the application. Figure 20 provides an example of TCS Output, where each TCS method follows the format <class name:method name:line number>. The most recent called method appears at the top and the least recent at the bottom.

```
function run_native_code(target_filename)
{
  var open_hook;
  var write_hook;

  open_hook = Interceptor.attach (Module.findExportByName ('
     libc.so', 'open'),
  {
    onEnter: function (args)
    {
      var value = Memory.readUtf8String(args[0]);
      if (typeof value !== 'undefined')
        if (value.indexOf(target_filename) !== -1)
          this._open_fileName = value;
    },
    onLeave: function (retval)
    {
      if (retval.toInt32() > 0)
        file_descriptor_array[retval.toInt32()] = this.
           _open_fileName;
    }
  });

  write_hook = Interceptor.attach (Module.findExportByName (
     'libc.so', 'write'),
  {
    onEnter: function (args)
    {
      this._write_fName = file_descriptor_array[args[0].
         toInt32()];
    },
    onLeave: function (retval)
    {
      if (retval.toInt32() > 0)
      {
        if (typeof this._write_fName !== 'undefined')
        {
          if (this._write_fName.indexOf(target_filename) !==
              -1)
          {
            var stack_trace = get_stacktrace();
            if (stack_trace.length > 2)
            {
              send(
              {
                from: '/sampler',
                name: '+update',
                payload:
                  STACKTRACE: stack_trace
              });
              open_hook.detach();
              write_hook.detach();
              }}}}}});}
```

Listing IV.3. Code Injector function `run_native_code`.

D3 (Interact): After the Integrator injects a hook (either native or virtual) into the

targeted application, the examiner must perform an action to trigger the re-writing of the targeted file. The interaction is application specific, but our research shows that merely backgrounding and resuming the targeted application will accomplish the objective.

D4 (Trace): Upon receipt of the TCS Output over Frida's messaging service, the Integrator correlates the TCS method call to the parsed smali tokens that ADSS S2 stored in the database `app_lib_db` during the Static Phase. The Integrator traces the code executed and forms the TCS Tree, a data structure that tracks objects (function parameters, and register-type pairs) as they enter and leave each TCS method. Figure 21 conceptually shows how the tree would appear. Additional method calls occur between each TCS method, denoted by f_call_N(). Based on our research, it is within the calls to f_call_N() that will yield a function that takes as input an object and begins the process to serialize the object into data written to the targeted file. Because the Android Virtual Machine's implementation mirrors the Java Virtual Machine, ADSS takes into consideration that objects as annotated in the smali code may be parent classes of the runtime object being serialized.

```
1  function get_stacktrace()
2  {
3    var trace_string = "";
4    if (Java.available)
5    {
6      Java.perform(function ()
7      {
8        var thread = Java.use('java.lang.Thread');
9        var trace = thread.currentThread().getStackTrace();
10       if (trace.length > 0)
11       {
12         for (var i = 0; i < trace.length; i++)
13           trace_string += 'TRACE:/' + trace[i].getFileName()
                 + ':' + trace[i].getMethodName() + '():' +
               trace[i].getLineNumber() + '\n';
14       }
15     });
16   }
17   return trace_string;
18 }
```

Listing IV.4. Function get_stack_trace().

The Integrator resolves the runtime type of these methods by injecting either a pre-

hook or post-hook to retrieve runtime information about the parameters of the hooked method. Void methods which do not return follow the pre-hook and post-hook method except without the `return` keyword. Within the hook, a JavaScript call to `obj.getClass().toString()` reveals the runtime type of object `obj`. Using this technique, the Integrator updates the TCS Tree to more accurately depict the TCS path executed after each successive hook.

In the case where ADSS needs to determine the type of a method call that is dependent on dynamically generated or local variables declared within the method scope, ADSS will convert the opcodes executed thus far into JavaScript code inside the virtual hook sent to the Code Injector.

```
VMStack.java:getThreadStackTrace() :-2
Thread.java:getStackTrace():1566
Posix.java:writeBytes():-2
Posix.java:write():273
BlockGuardOs.java:write() :319
IoBridge.java:write() :496
FileOutputStream.java:write() :316
app_class1:TCS_Func_212:61
app_class12:TCS_Func_191:61
app_class34:TCS_Func_14:255
app_class4:TCS_Func_88:129
app_class95:TCS_Func_19:13
app_class62:TCS_Func_21:98
app_class7:TCS_Func_76:13
FutureTask.java:run() :20
ScheduledThreadPoolExecutor.java:run():22
ThreadPoolExecutor.java:runWorker():33
ThreadPoolExecutor.java:run():67
Thread.java:run():71
```

Figure 20. Stack Trace Output.

D5 (Taint): Tainting and Tracing work in concert with each other, such that a method `invoke` is only traced if it's parameters contain a tainted object. If the

runtime type of a tainted register is unknown, meaning the static smali code shows a parent class, then the Integrator, as discussed in D4 Trace, can dynamically resolve the runtime type of the parent class invoking the method so that it can then be traced and tainted.

Outside of register tainting within method scope, ADSS taints data as it is written to a buffer and then dumped to the targeted file. It accomplishes this by tainting at the byte and object levels. If the first 20 bytes of the targeted file saved in D1 Integrator matches the first 20 bytes of the byte array that is being tracked in the TCS Tree prior to dumping to the targeted file, then the application objects being serialized to that buffer are associated with the targeted file. Furthermore, all the methods that have been identified as passing tainted registers can now filter their their data flows from the hashcode of the buffer object. This filtering displays application objects (ie data structures) and their associated bytes only if the bytes are written to the previously identified buffer. By tainting the memory buffer, this allows ADSS to monitor data written to the targeted file that may not be directly associated with the Main Object previously found.

Output: ADSS outputs three products for the examiner. The Final Hook List is a JavaScript list of the virtual hooks injected into the running application. With this list, the examiner can restart the application and inject all the hooks simultaneously without a need to continual interaction between each hook. The Main Object Schema is a tree structure that displays the types of sub objects dependent on the Main Object. The Hex-to-Object is a running list of pairs between the object runtime type and the hex representation from each virtual method hooked in the Final Hook List. It is from the Hex-to-Object output that the examiner can compare the bytes linked to data structures with the bytes of the targeted file that represents the File Format of the targeted file. The examiner can attribute any data not found in the

Hex-to-Object Output from ADSS but present in the targeted file as pre-meta-data.

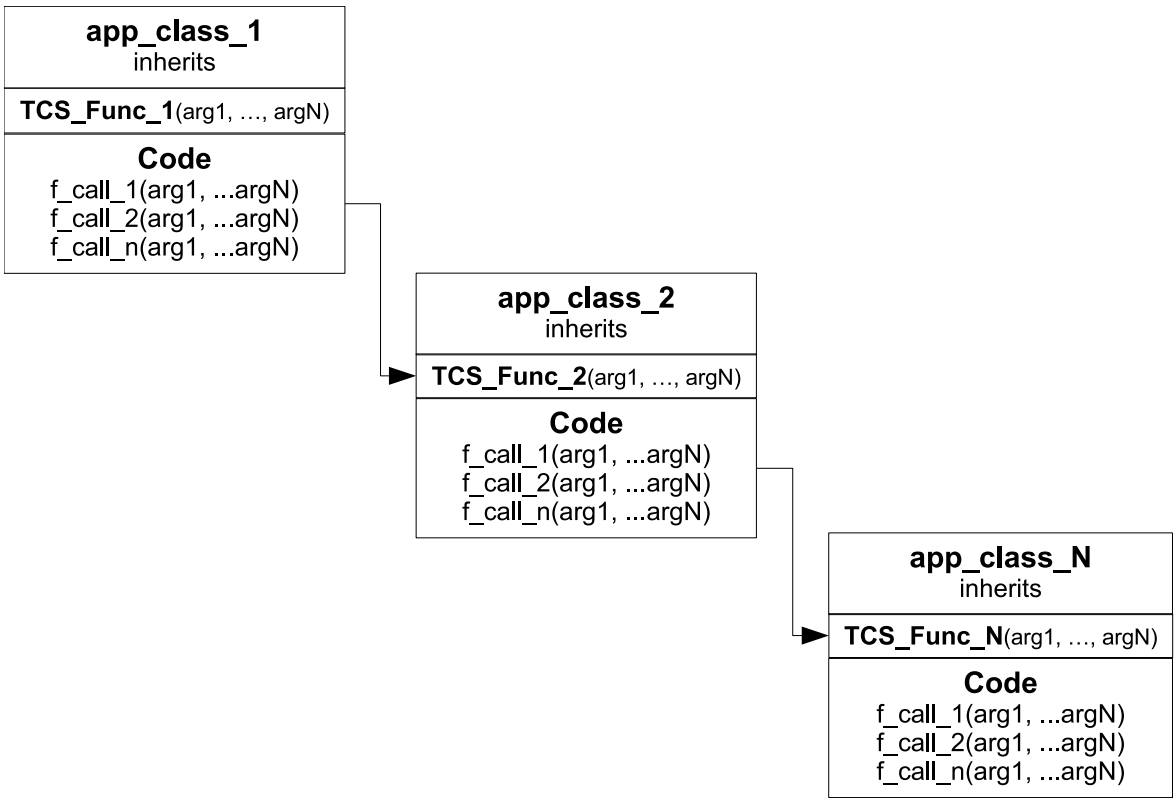

Figure 21. TCS Tree.

4.4 Summary

In summary, the Automated Data Structure Slayer (ADSS) applies traditional reverse engineering techniques toward reversing the file formats used by modern Android Applications. It automates this process so that minimal effort is required by the examiner. After identifying the file of interest, the examiner starts ADSS with the application and the file. The system processes the application statically and then uses a series of dynamic injection techniques to hook the methods called along the Thread Call Stack to associate data structures with the binary data forming the file of interest. The resulting output is the file format for the file selected.

The next chapter discusses the results from applying ADSS to Discord and Uber.

V. Evaluation

The Automated Data Structure Slayer (ADSS) is a tool that implements a new method to determine file formats of Android applications that store user data in unknown data formats. It conducts comprehensive static analysis on a targeted application, pulling out classes, methods, and other static data structures, and then dynamically interacts with the application to determine the data structures as they are converted to bytes and stored in a forensic file of interest. The core source code for ADSS is written in python and leverages open source technologies – APKTool, Frida, and built in Android tools to accomplish this objective.

This section demonstrates the results from applying ADSS on the popular Android application Uber (v4.208.10003) and Discord (v6.6.1). This section discusses identifying the paths executed, reveals the serializers and application objects, shows the serialized objects in their hex representation, displays the for each file, and reveals the final format respectively. Lastly, using the ADSS output, analyst can extract the evidentiary objects from files `realtime-demo_KEY_RIDER` and `STORE_MESSAGES_CACHE_V17` which were created on a different phone, the Galaxy MP running Uber and Discord applications.

5.1 Experimental Design

The evaluation experiments use the Nexus 6P device running Android (v6.0). Android was rooted using SuperSU (v2.8.2) [89] and the firmware recovery image provided by Team Win Recovery Project (TWRP) (v2.8.4) [90] to provide increased privilege and access to the internal files generated under secure the application folders. ADSS runs on Linux Mint (v 18.3) [91] and communicates to the device via the Android Debug Bridge (ADB). In addition Frida server (v7.8.6) [86] for Android

ARM64 and Uber (v3.154.2) is running on the device. This setup to remotely interact with an application via ADB is consistent with other dynamic analysis approaches, namely TaintDroid and Droit, with the exception of using an actual device versus an emulator.

5.2 Uber

Uber [92], a peer-to-peer ride-sharing Android application, is currently listed at having over 100M downloads as of June 2018. It allows people to work as drivers and for others to use the application to request rides to their destinations. It is available in over 600 US cities and the company has a valuation worth over 60 Billion dollars as of 2017 [93].

In this experiment, we created an Uber user and used the Uber application to ride from one location to another. Manual inspection showed that Uber generated files under folder `/data/data/com.ubercab/files/`. Running the standard Linux `file` on file `realtime-demo_KEY_RIDER` revealed that it was of an unknown data format; this file then became the file of interest for this experiment. `realtime-demo_KEY_RIDER` stores profile information about the Uber user, such as email address, username, phone number, and the full name.

To begin the experiment, ADSS receives the file of interest and the Uber application as input. It then parses, patches, and prepares Uber in accordance with ADSS Static Step 1 and 2. There were approx 70,000 smali files generated from the disassembly and decompression process. Analysis shows the Uber application relies on native and virtual machine code to execute. It is very complex with multiple levels of inheritance and the application employs code obfuscation techniques that rename variables, methods, and classes. Furthermore, Uber uses a third-party library, Kryo, to store application data.

Figure 22 shows the thread call stack that is executed before Uber writes to file `realtime-demo_KEY_RIDER`.

```
VMStack.java:getThreadStackTrace():-2
Thread.java:getStackTrace():580
Posix.java:writeBytes():-2
Posix.java:write():271
BlockGuardOs.java:write():313
IoBridge.java:write():493
FileOutputStream.java:write():186
smali_classes4/alhx$1.smali:write():78
smali_classes4/alhy.smali:close():229
smali_classes8/ddh.smali:a():36
smali_classes8/dcr.smali:e():277
smali_classes8/dcr.smali:a():34
smali_classes8/dcr$1.smali:a():102
smali_classes4/akng.smali:a():34
smali/akil.smali:b():1634
smali_classes4/akns.smali:run():64
smali_classes4/akjf$1.smali:run():138
smali_classes4/akzy.smali:run():59
smali_classes4/akzy.smali:call():51
FutureTask.java:run():237
ScheduledThreadPoolExecutor.java:run():269
ThreadPoolExecutor.java:runWorker():1113
ThreadPoolExecutor.java:run():588
Thread.java:run():818
```

Figure 22. Uber Thread Call Stack.

ADSS parses in the thread call stack and develops a hybrid dynamic and static data structure to illuminate the path Uber executes consistent with the thread call stack. The tree is too large to re-create in this paper, but Figure 23 shows the salient elements (D1 to D9) of the path that Uber takes to write data to the file `realtime-demo_KEY_RIDER`. The steps D1, D2, and D5 are the dynamic points where ADSS injects code to determine the runtime type of the object making the function call at those junctions. D1 reveals the main type of the object as `com.uber.model.core.`

generated.rtapi.models.rider.AutoValue_Rider. Because this is the first custom object which ADSS found that is stored in the targeted file, when juncture D1 passes this object as an argument, then ADSS adjusts its filter to see all traffic which is written to the output buffer of type com.esotericsoftware.kryo.io.Output. A curious analyst would inspect this output buffer and identify that it extends java.io.OutputStream by adding additional functionality to the class. Now data can be identified as it is passed into D3 and D4 because ADSS is filtering on the output buffer. At juncture D3, CompatibleFieldsSerializer.write() writes the number of fields present in object com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider as the second byte to file realtime-demo_KEY_RIDER by making a call to D3 in this case there are 35 fields (or 0x23). Next, Uber writes each field name com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider as ASCII encoded strings at junction D4 – this forms the header. Junctures D3 and D4 write the header of file realtime-demo_KEY_RIDER. A runtime type of null means there is no data associated with the field and Uber writes (0x00).

Junctures D5 - D9 are used to write the body of file realtime-demo_KEY_RIDER. Each field is serialized into a chunk of data as it passes through D5 and is dynamically resolved to go through D7, D8, or D9. If the field is an object, then it goes through D7. The primitive types, in this case java.lang.Boolean and java.lang.Integer pass through junctures D8 and D9 as arguments respectively. After each field of object com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider is serialized, juncture D6 occurs, which appends the size of bytes to be written at the end of the written chunk of data. This is used by Uber's own parser that reads in the chunks of each file and can determine how long a chunk of data is to be expected.

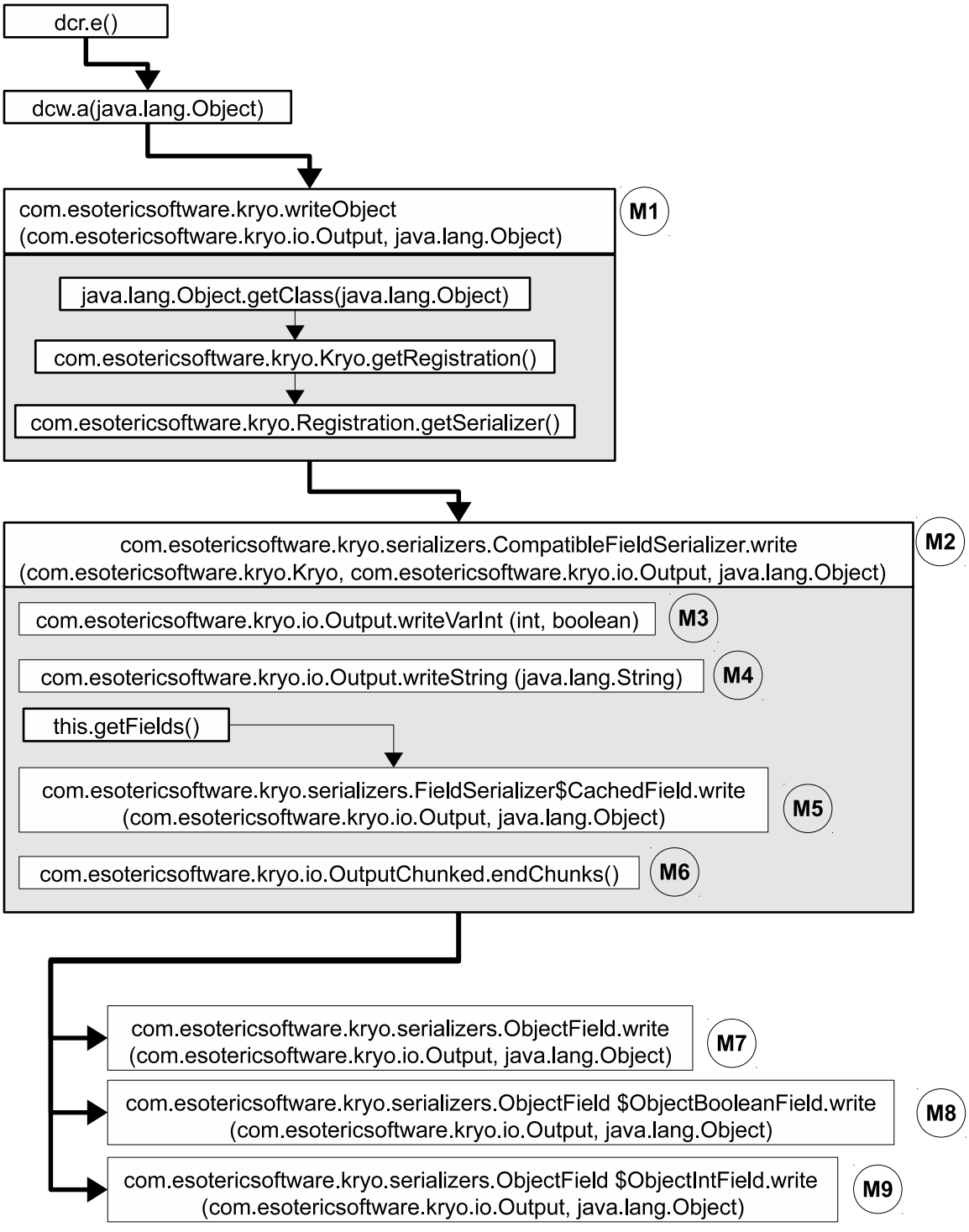

Figure 23. Uber Call Tree.

Table 3. Uber Results.

Target File: /com.ubercab/files/realtime-demo_KEY_RIDER			
Main Runtime Type: com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider			
Name	Runtime Type	Serializer	Offset
claimedMobile	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00001E01
creditBalances	lnd	com.esotericsoftware.kryo.serializers.ObjectField	0x00001E04
email	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00001E09
firstName	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00001F0E
hasConfirmedMobile	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002006
hasConfirmedMobileStatus	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x0000200A
hasNoPassword	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002100
hasToOptInSmsNotifications	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002104
hashCode	java.lang.Integer	com.esotericsoftware.kryo.serializers.ObjectField $ObjectIntField	0x00002108
hashCode$Memoized	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField $ObjectBooleanField	0x0000210B
isAdmin	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x0000210E
isTeen	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002202
lastExpenseInfo	com.uber.model.core.generated.rtapi.models.expenseinfo.AutoValue_ExpenseInfo	com.esotericsoftware.kryo.serializers.ObjectField	0x00002206
lastExpenseMemo	com.uber.model.core.generated.rtapi.models.rider.ExpenseMemo	com.esotericsoftware.kryo.serializers.ObjectField	0x00002E0B
lastName	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00002E0E
lastSelectedPaymentGoogleWalletUUID	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002F08
lastSelectedPaymentProfileIsGoogleWallet	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00002F0B
lastSelectedPaymentProfileUUID	com.uber.model.core.generated.rtapi.models.payment.AutoValue_PaymentProfileUuid	com.esotericsoftware.kryo.serializers.ObjectField	0x00003000
meta	com.uber.model.core.generated.rtapi.models.object.AutoValue_Meta	com.esotericsoftware.kryo.serializers.ObjectField	0x00003A01
mobileCountryIso2	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00004B0F
mobileDigits	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00004C04
pictureUrl	com.uber.model.core.generated.rtapi.models.rider.AutoValue_URL	com.esotericsoftware.kryo.serializers.ObjectField	0x00004D02
profileType	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00005701
profiles	lnd	com.esotericsoftware.kryo.serializers.ObjectField	0x00005704
promotion	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00005709
rating	java.lang.Double	com.esotericsoftware.kryo.serializers.ObjectField	0x0000570C
recentFareSplitters	lnd	com.esotericsoftware.kryo.serializers.ObjectField	0x00005807
referralCode	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x0000580C
referralUrl	com.uber.model.core.generated.rtapi.models.rider.AutoValue_URL	com.esotericsoftware.kryo.serializers.ObjectField	0x0000590B
role	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00005D01
thirdPartyIdentities	lnf	com.esotericsoftware.kryo.serializers.ObjectField	0x00005D0A
toString	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00005D0F
tripBalances	lnd	com.esotericsoftware.kryo.serializers.ObjectField	0x00005E02
userType	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00005E05
uuid	com.uber.model.core.generated.rtapi.models.rider.AutoValue_RiderUuid	com.esotericsoftware.kryo.serializers.ObjectField	0x00005E09

By placing a dynamic hook at D3 and displaying the buffer of data written before and after the call, ADSS associates the new bytes added to the buffer with a serialized field. The header of file `realtime-demo_KEY_RIDER` in hex is represented in Figure while the body portion of the file is represented in Figure 25 and Figure 26.

Table 3 summarizes the runtime types of the object and sub-objects stored in file `realtime-demo_KEY_RIDER` and identifies the starting hex offset of each field as it appears in Figure 25 and Figure 26.

It is important to note that after ADSS inserts the dynamic hook, the analyst will have to do an action with the application to trigger the re-writing of the targeted file again. From `0x0000000` to `0x00001D0` is the header of the file. The header contains the concatenation of the sub object field names. The second byte, at offset 0x00000001 stores the number of fields that are written in the file. In this case, `23` shows that there are 35 fields and their bytes stored in this field. The header is a list of strings separated by a non-ascii character (in bold). Each string is the name of a field. For example, 0x00000602 to 0x00000803 holds the name for field `hasConfirmedMobileStatus`.

The body, the remaining part of the file, contains the serialized objects in the order identified in the header. The bold hex store the length of the next serialized object. This is a meta data for the application to easily read in the next serialized object; it indicates the length of the next object appearing in the file, which is in green hex. For example, starting at offset `0x000100D`, `65 6D 61 69 6C` is the header name `email`. This is the field name listed in the header. The byte representation of the value stored in header `email` is located at offset `0x00001E0A` as `6A 33 61 6D 65 73 6F 6E 40 67 6D 61 69 6C 2E 63 6F`[1], which is 20 bytes long. The meta field that precedes this is `00 14`, which is 20 in decimal, the length of the data.

[1]actual email address masked

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000000	**01**	**23**	63	6C	61	69	6D	65	64	4D	6F	62	69	6C	**E5**	63
0000010	72	65	64	69	74	42	61	6C	61	6E	63	65	**F3**	65	6D	61
0000020	69	**EC**	66	69	72	73	74	4E	61	6D	**E5**	68	61	73	43	6F
0000030	6E	66	69	72	6D	65	64	4D	6F	62	69	6C	**E5**	68	61	73
0000040	43	6F	6E	66	69	72	6D	65	64	4D	6F	62	69	6C	65	53
0000050	74	61	74	75	**F3**	68	61	73	4E	6F	50	61	73	73	77	6F
0000060	72	**E4**	68	61	73	54	6F	4F	70	74	49	6E	53	6D	73	4E
0000070	6F	74	69	66	69	63	61	74	69	6F	6E	**F3**	68	61	73	68
0000080	43	6F	64	**E5**	68	61	73	68	43	6F	64	65	24	4D	65	6D
0000090	6F	69	7A	65	**E4**	69	73	41	64	6D	69	**EE**	69	73	54	65
00000A0	65	**EE**	6C	61	73	74	45	78	70	65	6E	73	65	49	6E	66
00000B0	**EF**	6C	61	73	74	45	78	70	65	6E	73	65	4D	65	6D	**EF**
00000C0	6C	61	73	74	4E	61	6D	E5	6C	61	73	74	53	65	6C	65
00000D0	63	74	65	64	50	61	79	6D	65	6E	74	47	6F	6F	67	6C
00000E0	65	57	61	6C	6C	65	74	55	55	49	C4	6C	61	73	74	53
00000F0	65	6C	65	63	74	65	64	50	61	79	6D	65	6E	74	50	72
0000100	6F	66	69	6C	65	49	73	47	6F	6F	67	6C	65	57	61	6C
0000110	6C	65	**F4**	6C	61	73	74	53	65	6C	65	63	74	65	64	50
0000120	61	79	6D	65	6E	74	50	72	6F	66	69	6C	65	55	55	49
0000130	**C4**	6D	65	74	E1	6D	6F	62	69	6C	65	43	6F	75	6E	74
0000140	72	79	49	73	6F	**B2**	6D	6F	62	69	6C	65	44	69	67	69
0000150	74	**F3**	70	69	63	74	75	72	65	55	72	**EC**	70	72	6F	66
0000160	69	6C	65	54	79	70	**E5**	70	72	6F	66	69	6C	65	**F3**	70
0000170	72	6F	6D	6F	74	69	6F	**EE**	72	61	74	69	6E	**E7**	72	65
0000180	63	65	6E	74	46	61	72	65	53	70	6C	69	74	74	65	72
0000190	**F3**	72	65	66	65	72	72	61	6C	43	6F	64	**E5**	72	65	66
00001A0	65	72	72	61	6C	55	72	**EC**	72	6F	6C	**E5**	74	68	69	72
00001B0	64	50	61	72	74	79	49	64	65	6E	74	69	74	69	65	**F3**
00001C0	74	6F	53	74	72	69	6E	**E7**	74	72	69	70	42	61	6C	61
00001D0	6E	63	65	F3	75	73	65	72	54	79	70	**E5**	75	75	69	**E4**

Figure 24. Header of file `realtime-demo_KEY_RIDER`.

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00001E0	**01**	00	**00**	**03**	01	01	00	**00**	**13**	01	6A	33	61	6D	65	73
00001F0	6F	6E	40	67	6D	61	69	6C	2E	63	6F	ED	**00**	**06**	01	4A
0000200	61	63	6F	E2	**00**	**02**	01	01	**00**	**04**	01	59	65	F3	**00**	**02**
0000210	01	00	**00**	**02**	01	00	**00**	**01**	00	**00**	**01**	00	**00**	**02**	01	00
0000220	**00**	**01**	00	**00**	**C3**	**01**	01	00	CD	01	63	6F	6D	2E	75	62
0000230	65	72	2E	6D	6F	64	65	6C	2E	63	6F	72	65	2E	67	65
0000240	6E	65	72	61	74	65	64	2E	72	74	61	70	69	2E	6D	6F
0000250	64	65	6C	73	2E	65	78	70	65	6E	73	65	69	6E	66	6F
0000260	2E	41	75	74	6F	56	61	6C	75	65	5F	45	78	70	65	6E
0000270	73	65	49	6E	66	6F	01	08	61	6E	6E	6F	74	61	74	69
0000280	6F	6E	45	72	72	6F	F2	62	75	73	69	6E	65	73	73	54
0000290	72	69	F0	63	6F	64	E5	65	78	70	65	6E	73	65	54	72
00002A0	69	F0	68	61	73	68	43	6F	64	E5	68	61	73	68	43	6F
00002B0	64	65	24	4D	65	6D	6F	69	7A	65	E4	6D	65	6D	EF	74
00002C0	6F	53	74	72	69	6E	E7	01	00	00	02	01	00	00	05	01
00002D0	6E	75	6C	EC	00	02	01	00	00	01	00	00	01	00	00	05
00002E0	01	6E	75	6C	EC	00	01	00	00	**00**	**01**	00	**00**	**08**	01	4A
00002F0	61	6D	65	73	6F	EE	**00**	**01**	00	**00**	**02**	01	00	**00**	**9E**	**01**
0000300	01	01	D0	01	63	6F	6D	2E	75	62	65	72	2E	6D	6F	64
0000310	65	6C	2E	63	6F	72	65	2E	67	65	6E	65	72	61	74	65
0000320	64	2E	72	74	61	70	69	2E	6D	6F	64	65	6C	73	2E	70
0000330	61	79	6D	65	6E	74	2E	41	75	74	6F	56	61	6C	75	65
0000340	5F	50	61	79	6D	65	6E	74	50	72	6F	66	69	6C	65	55
0000350	75	69	64	01	03	67	65	F4	68	61	73	68	43	6F	64	E5
0000360	68	61	73	68	43	6F	64	65	24	4D	65	6D	6F	69	7A	65
0000370	E4	25	01	64	64	37	30	31	37	62	38	2D	36	35	34	62
0000380	2D	34	31	39	39	2D	39	34	65	61	2D	31	39	38	63	38
0000390	36	35	37	64	38	65	E3	00	01	00	00	01	00	00	**00**	**9C**
00003A0	**02**	01	02	C1	01	63	6F	6D	2E	75	62	65	72	2E	6D	6F
00003B0	64	65	6C	2E	63	6F	72	65	2E	67	65	6E	65	72	61	74
00003C0	65	64	2E	72	74	61	70	69	2E	6D	6F	64	65	6C	73	2E
00003D0	6F	62	6A	65	63	74	2E	41	75	74	6F	56	61	6C	75	65
00003E0	5F	4D	65	74	61	01	05	68	61	73	68	43	6F	64	E5	68
00003F0	61	73	68	43	6F	64	65	24	4D	65	6D	6F	69	7A	65	E4
0000400	6C	61	73	74	4D	6F	64	69	66	69	65	64	54	69	6D	65
0000410	4D	F3	6F	72	69	67	69	6E	54	69	6D	65	4D	F3	74	6F
0000420	53	74	72	69	6E	E7	01	00	00	01	00	00	77	01	03	C6
0000430	01	63	6F	6D	2E	75	62	65	72	2E	6D	6F	64	65	6C	2E
0000440	63	6F	72	65	2E	67	65	6E	65	72	61	74	65	64	2E	72
0000450	74	61	70	69	2E	6D	6F	64	65	6C	73	2E	74	73	2E	41
0000460	75	74	6F	56	61	6C	75	65	5F	54	69	6D	65	73	74	61
0000470	6D	70	49	6E	4D	73	01	03	67	65	F4	68	61	73	68	43
0000480	6F	64	E5	68	61	73	68	43	6F	64	65	24	4D	65	6D	6F

Figure 25. Hex representation file `realtime-demo_KEY_RIDER`.

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000490	69	7A	65	E4	08	42	76	31	C8	B4	A6	B0	00	00	01	00
00004A0	00	01	00	00	00	13	01	03	01	08	42	76	31	C8	B4	A3
00004B0	10	00	00	01	00	00	01	00	00	00	01	00	00	**00**	**03**	01
00004C0	55	D3	**00**	**0B**	01	33	30	31	34	30	36	31	30	38	39	**00**
00004D0	**99**	**01**	01	04	63	6F	6D	2E	75	62	65	72	2E	6D	6F	64
00004E0	65	6C	2E	63	6F	72	65	2E	67	65	6E	65	72	61	74	65
00004F0	64	2E	72	74	61	70	69	2E	6D	6F	64	65	6C	73	2E	72
0000500	69	64	65	72	2E	41	75	74	6F	56	61	6C	75	65	5F	55
0000510	52	CC	01	03	67	65	F4	68	61	73	68	43	6F	64	E5	68
0000520	61	73	68	43	6F	64	65	24	4D	65	6D	6F	69	7A	65	E4
0000530	33	01	68	74	74	70	73	3A	2F	2F	64	31	77	32	70	6F
0000540	69	72	74	62	33	61	73	39	2E	63	6C	6F	75	64	66	72
0000550	6F	6E	74	2E	6E	65	74	2F	64	65	66	61	75	6C	74	2E
0000560	6A	70	65	E7	00	01	00	00	01	00	00	**00**	**05**	01	75	62
0000570	65	F2	**00**	**03**	01	01	00	**00**	**01**	00	**00**	**09**	01	40	14	00
0000580	00	00	00	00	00	**00**	**03**	01	01	00	**00**	**0D**	01	6A	61	63
0000590	6F	62	6A	36	37	34	35	75	E9	**00**	**34**	01	04	01	29	01
00005A0	68	74	74	70	73	3A	2F	2F	77	77	77	2E	75	62	65	72
00005B0	2E	63	6F	6D	2F	69	6E	76	69	74	65	2F	6A	61	63	6F
00005C0	62	6A	36	37	34	35	75	E9	00	01	00	00	01	00	00	**00**
00005D0	**07**	01	63	6C	69	65	6E	F4	**00**	**03**	01	01	00	**00**	**01**	00
00005E0	**00**	**01**	00	**00**	**01**	00	**00**	**93**	**01**	01	05	C5	01	63	6F	6D
00005F0	2E	75	62	65	72	2E	6D	6F	64	65	6C	2E	63	6F	72	65
0000600	2E	67	65	6E	65	72	61	74	65	64	2E	72	74	61	70	69
0000610	2E	6D	6F	64	65	6C	73	2E	72	69	64	65	72	2E	41	75
0000620	74	6F	56	61	6C	75	65	5F	52	69	64	65	72	55	75	69
0000630	64	01	03	67	65	F4	68	61	73	68	43	6F	64	E5	68	61
0000640	73	68	43	6F	64	65	24	4D	65	6D	6F	69	7A	65	E4	25
0000650	01	31	65	31	37	62	34	38	32	2D	66	36	64	63	2D	34
0000660	38	33	64	2D	61	34	36	66	2D	37	63	31	65	31	66	62
0000670	39	66	38	34	B3	00	01	00	00	01	00	00	**00**			

Figure 26. Hex representation file `realtime-demo_KEY_RIDER`.

To further aid in deciphering the file format, ADSS produces a schema of the targeted file, located in Figure 27. This is a tree structure which shows the types and subtypes of the object `com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider` which form the backbone of the file `realtime-demo_KEY_RIDER`.

```
com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider
|
+-> claimedMobile (java.lang.String)
+-> creditBalances (lnd)
+-> email (java.lang.String)
+-> firstName (java.lang.String)
+-> hasConfirmedMobile (java.lang.Boolean)
+-> hasConfirmedMobileStatus (java.lang.String)
+-> hasNoPassword (java.lang.Boolean)
+-> hasToOptInSmsNotifications (java.lang.Boolean)
+-> hashCode$Memoized (java.lang.Boolean)
+-> hashCode (java.lang.Integer)
+-> isAdmin (java.lang.Boolean)
+-> isTeen (java.lang.Boolean)
+-> lastExpenseInfo (com.uber.model.core.generated.rtapi.models.expenseinfo.AutoValue_
ExpenseInfo)
|   |
|   +-> annotationError (java.lang.String)
|   +-> businessTrip (java.lang.String)
|   +-> code (java.lang.String)
|   +-> expenseTrip (java.lang.String)
|   +-> hashCode (java.lang.String)
|   +-> hashCode$Memoized (java.lang.String)
|   +-> memo (java.lang.String)
|   +-> toString (java.lang.String)
|
+-> lastExpenseMemo (com.uber.model.core.generated.rtapi.models.rider.ExpenseMemo)
+-> lastName (java.lang.String)
+-> lastSelectedPaymentGoogleWalletUUID (java.lang.Boolean)
+-> lastSelectedPaymentProfileIsGoogleWallet (java.lang.Boolean)
+-> lastSelectedPaymentProfileUUID (com.uber.model.core.generated.rtapi.models.
payment.AutoValue_PaymentProfileUuid)
|   |
|   +-> get (java.lang.String)
|   +-> hashCode (java.lang.String)
|   +-> hashCode$Memoized (java.lang.String)
|
+-> meta (com.uber.model.core.generated.rtapi.models.object.AutoValue_Meta)
|   |
|   +-> hashCode  (java.lang.String)
|   +-> hashCode$Memoized  (java.lang.String)
|   +-> lastModifiedTimeMs  (java.lang.String)
|   +-> originTimeMs  (java.lang.String)
|   +-> toString  (java.lang.String)
|
+-> mobileCountryIso2 (java.lang.String)
+-> mobileDigits (java.lang.String)
|
|
```

Figure 27. Schema from Rider.

```
+-> pictureUrl (com.uber.model.core.generated.rtapi.models.rider.AutoValue_URL)
|   |
|   +-> get (java.lang.String)
|   +-> hashCode (java.lang.String)
|   +-> hashCode$Memoized (java.lang.String)
|
+-> profileType (java.lang.String)
+-> profiles (lnd)
+-> promotion (java.lang.String)
+-> rating (java.lang.Double)
+-> recentFareSplitters (lnd)
+-> referralCode (java.lang.String)
+-> referralUrl (com.uber.model.core.generated.rtapi.models.rider.AutoValue_URL)
|   |
|   +-> get (java.lang.String)
|   +-> hashCode (java.lang.String)
|   +-> hashCode$Memoized (java.lang.String)
|
+-> role (java.lang.String)
+-> thirdPartyIdentities (lnf)
+-> toString (java.lang.String)
+-> tripBalances (lnd)
+-> userType (java.lang.String)
+-> uuid (com.uber.model.core.generated.rtapi.models.rider.AutoValue_RiderUuid)
        |
        +-> get (java.lang.String)
        +-> hashCode (java.lang.String)
        +-> hashCode$Memoized (java.lang.String)
```

Figure 28. Schema from Rider (cont'd).

Figure 29 shows the file format for `realtime-demo_KEY_RIDER`.

Tested the validity of this file format consists of installing the same version of Uber on another device, the Samsung Galaxy MP running Android (v4.3). We created another user account and rode in an Uber car to a new destination. As expected, the application generated a similar file and with our knowledge of the file format we were able to develop a parser to extract the evidentiary data structures from the file `realtime-demo_KEY_RIDER`. The parser is located in Appendix C.

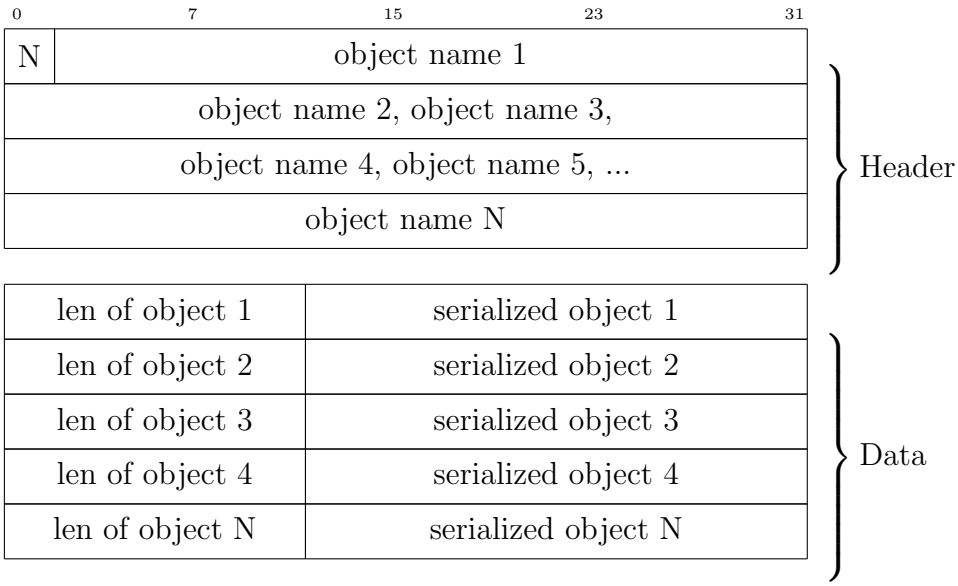

Figure 29. Format of file `realtime-demo_KEY_RIDER`.

5.3 Discord

Discord [94], a communication Android application designed for gamming communities, is currently listed at having over 10M downloads as of June 2018. It allows people to chat, talk, and video message, and send pictures to one another. It has over 45 Million registered users and the company has a valuation worth over 750 Million dollars as of 2017 [95].

In this experiment, we created two users and signed each user in on a different phone - Pixel XL and Nexus 6P. The Nexus 6P was the rooted device running Android (v6.0), Frida server (v7.8.6), and Discord (v6.4.7). The different users sent back and for text messages, pictures, and audio through Voice over IP. Manual inspection of the Nexus 6P revealed that Discord generated files under folder `/data/data/com.discord/files/`. Running the standard Linux `file` on file

STORE_MESSAGES_CACHE_V17 showed that it was of an unknown data format; this file then became the file of interest for this experiment. STORE_MESSAGES_CACHE_V-17 stores the entire chat log history for the Discord user, profile details, and other evidentiary user information.

To begin the experiment, ADSS receives the file of interest and the Discord application as input. It then parses, patches, and prepares Discord in accordance with with ADSS Static Step 1 and 2. There were approx 15,000 smali files generated from the disassembly and decompression process. Analysis shows the Discord application relies on native and virtual machine code to execute. Manual inspection identifies that this application is not as complex as the Uber application but it still does use polymorphism and standard code obfuscation techniques that rename variables, methods, and classes. Furthermore, Discord also uses the third-party library, Kryo, to store application data, albeit in a different file format serializer.

Figure 30 shows the thread call stack that ADSS generates from hooking the native standard C `libc.so` `write()` and `open()` functions as Discord writes data to the targeted file STORE_MESSAGES_CACHE_V17.

Similarly with Uber, ADSS parses in the thread call stack and develops a hybrid dynamic and static data structure to illuminate the path Discord executes consistent with the thread call stack. Again, the tree is too large to re-create in this paper, but 31 shows the final path Discord takes to write data to file STORE_MESSAGES_CACHE_V-17. The steps D1, D2, and D3 are the dynamic points where ADSS injects code to determine the runtime type of the object making the function call at those junctions. D1 reveals the main type of the object as `com.discord.models.domain.ModelMessage`. D2 shows that it serializes the main type's fields. D3 identifies the runtime type of the serialized fields and the associated serializers. Table 5 summarizes the runtime types of the object and sub-objects stored in file STORE_MESSAGES_CACHE_V17.

```
VMStack.java:getThreadStackTrace():-2
Thread.java:getStackTrace():580
Posix.java:writeBytes():-2
Posix.java:write():271
BlockGuardOs.java:write():313
IoBridge.java:write():493
FileOutputStream.java:write():186
com/esotericsoftware/kryo/io/Output.smali:flush():185
com/esotericsoftware/kryo/io/Output.smali:close():196
smali/kotlin/d/a.smali:a():65
smali/com/discord/utilities/persister/Persister.smali:
        persist():14
smali/com/discord/utilities/persister/Persister$Companion$
   persistAll$1$1.smali:invoke():250
smali/com/discord/utilities/persister/Persister$Companion$
   persistAll$1$1.smali:invoke():196
smali/com/discord/app/j.smali:call():-1
smali/rx/internal/util/b.smali:onNext():39
smali/rx/observers/b.smali:onNext():134
smali/rx/internal/a/ai$a.smali:call():224
smali/rx/internal/c/b$a$1.smali:call():172
smali/rx/internal/c/j.smali:run():55
Executors.java:call():423
FutureTask.java:run():237
ScheduledThreadPoolExecutor.java:run():269
ThreadPoolExecutor.java:runWorker():1113
ThreadPoolExecutor.java:run():588
Thread.java:run():818
```

Figure 30. Discord Thread Call Stack.

A runtime type of `null` means there is no data associated with the field.

By placing a dynamic hook at D3 and displaying the buffer of data written before and after the call, ADSS associates the new bytes added to the buffer with a serialized field. It is important to note that after ADSS inserts the dynamic hook, the analyst will have do an action with application to trigger the re-writing of the targeted file. With Discord, this action is sending or receiving a text message. Immediately before the hook is injected, ADSS deletes the contents of the targeted file

to ensure that when the application writes it is not merely an appendation of data. Figure 32, Figure 33, Figure 34, and Figure 35 show the hex representation for file `STORE_MESSAGES_CACHE_V17`. This file is the result of of serializing several Discord application objects, nested objects, and Java type objects which hold forensic data used by the Discord application, to include chat logs, pictures, and profile details about the user. The bold and underlined two bytes (ie `01 02`) are the start of a `com.discord.models.domain.ModelMessage` object. The green hex following correspond to the byte presentation of `com.discord.models.domain.ModelMessage` object. This particular file is storing 11 such objects. It is worth noting that at offset `0x00000308` associates the two byte ID with a string representing the fully qualified name of the object being represented. Based on inspection, the future serialized objects only use the two byte ID to show the start of the same object.

This ID to string association is a pattern which continues throughout the file as this section will continue to highlight. The two byte fields associated with an object are listed in Table 4. It is by the hook to `com.esotericsoftware.kryo.Kryo.writeClass()` that the two byte field can be seen written in the buffer preceding the object serialization.

To further aid in deciphering the file format, ADSS produces schema of the tar-

Table 4. Discord Pre-meta class identifier table.

Pre-Meta	Runtime Type
01 00	java.util.HashMap
01 01	java.util.ArrayList
01 02	com.discord.models.domain.ModelMessage
01 03	com.discord.models.domain.ModelUser
01 04	java.util.concurrent.atomic.AtomicReference
01 05	java.util.LinkedHashMap
01 06	com.discord.models.domain.ModelMessageReaction
01 07	com.discord.models.domain.ModelMessageReaction$Emoji
01 08	com.discord.models.domain.ModelMessageAttachment
01 09	com.discord.models.domain.ModelMessage$Call

geted file, located in Figure 36. This is a tree structure which shows the types and subtypes of the object `com.discord.models.domain.ModelMessage` which form the backbone of the file `STORE_MESSAGES_CACHE_V17`.

There are 11 objects of file `STORE_MESSAGES_CACHE_V17`, brevity details on only one presented at offset 00003C09. Figure 37 shows the hex representation of the third `com.discord.models.domain.ModelMessage` object. The byte representation of each field alternates in color between green and blue. In addition, if the field is an object with pre-meta class identifier, then the two byte field is highlighted and is in the same color as the rest of the bytes that compose that particular field. For example, offset 0x000040B starts the serialization of object, which is type `com.discord.models.domain.ModelMessage$Call`.

With this file format, we then installed and used Discord on another device, the Samsung Galaxy MP running Android (v4.3). We used another user account on the Samsun Galaxy MP to communicate over Discord. As expected, the application generated a similar file and with our knowledge of the file format we were able to a parser to extract the evidentiary data structures from the file `STORE_MESSAGES_CACHE_V17`. The parser is located in Appendix D.

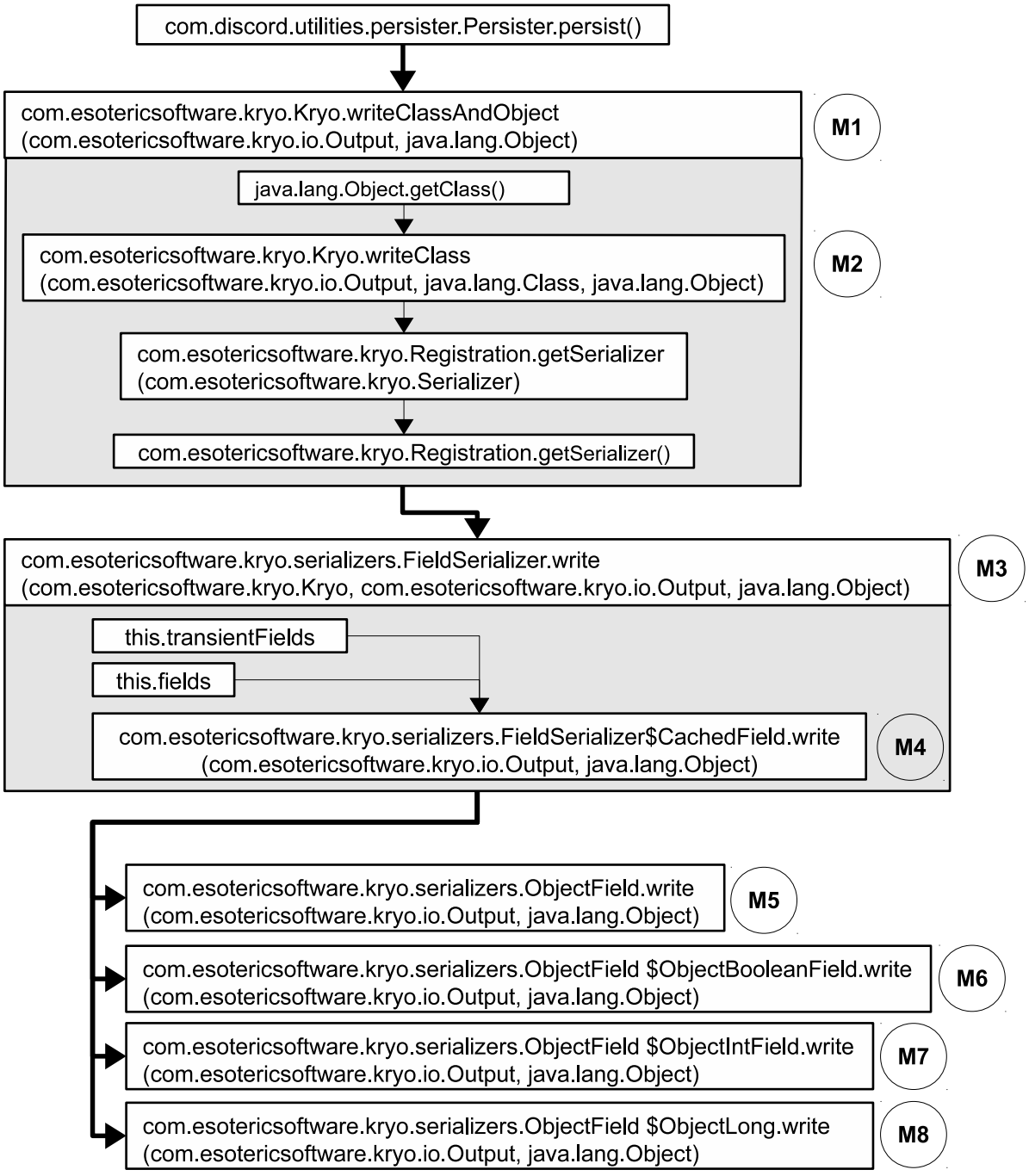

Figure 31. Discord Call Tree.

Table 5. Discord Results.

Target File: /com.discord/files/STORE_MESSAGES_CACHE_V17			
Main Runtime Type: com.discord.models.domain.ModelMessage			
Name	Runtime Type	Serializer	Offset
activity	com.discord.models.domain.ModelMessage.$Activity	com.esotericsoftware.kryo.serializers.ObjectField	0x00003C0B
application	com.discord.models.domain.ModelMessage$Application	com.esotericsoftware.kryo.serializers.ObjectField	0x00003C0D
attachments	java.util.ArrayList	com.esotericsoftware.kryo.serializers.ObjectField	0x00003C0E
author	com.discord.models.domain.ModelUser	com.esotericsoftware.kryo.serializers.ObjectField	0x00003D02
call	com.discord.models.domain.ModelMessage$Call	com.esotericsoftware.kryo.serializers.ObjectField	0x0000400B
channelId	java.lang.Long	com.esotericsoftware.kryo.serializers.ObjectField $ObjectLongField	0x00004704
content	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x0000470D
editedTimestamp	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x0000470F
editedTimestampMilliseconds	java.util.concurrent.atomic.AtomicReference	com.esotericsoftware.kryo.serializers.ObjectField	0x00004800
embeds	java.util.ArrayList	com.esotericsoftware.kryo.serializers.ObjectField	0x00004804
id	java.lang.Long	com.esotericsoftware.kryo.serializers.ObjectField $ObjectLongField	0x00004808
mentionEveryone	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField $ObjectBooleanField	0x00004901
mentionRoles	java.util.ArrayList	com.esotericsoftware.kryo.serializers.ObjectField	0x00004902
mentions	java.util.ArrayList	com.esotericsoftware.kryo.serializers.ObjectField	0x00004906
nonce	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00004D04
pinned	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00004D05
reactions	java.util.LinkedHashMap	com.esotericsoftware.kryo.serializers.ObjectField	0x00004D07
timestamp	java.lang.String	com.esotericsoftware.kryo.serializers.ObjectField	0x00004D08
timestampMilliseconds	java.util.concurrent.atomic.AtomicReference	com.esotericsoftware.kryo.serializers.ObjectField	0x00004F0A
tts	java.lang.Boolean	com.esotericsoftware.kryo.serializers.ObjectField	0x00005004
type	java.lang.Integer	com.esotericsoftware.kryo.serializers.ObjectField	0x00005005
webhookId	java.lang.Long	com.esotericsoftware.kryo.serializers.ObjectField	0x00005006

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000000	01	00	92	6A	61	76	61	2E	75	74	69	6C	2E	48	61	73
0000010	68	4D	61	70	01	01	09	80	80	90	E0	88	8E	F9	9E	0C
0000020	01	01	94	6A	61	76	61	2E	75	74	69	6C	2E	41	72	72
0000030	61	79	4C	69	73	74	01	09	**01**	**02**	**A7**	**63**	**6F**	**6D**	**2E**	**64**
0000040	**69**	**73**	**63**	**6F**	**72**	**64**	**2E**	**6D**	**6F**	**64**	**65**	**6C**	**73**	**2E**	**64**	**6F**
0000050	**6D**	**61**	**69**	**6E**	**2E**	**4D**	**6F**	**64**	**65**	**6C**	**4D**	**65**	**73**	**73**	**61**	**67**
0000060	**65**	01	00	00	01	01	01	00	01	03	A4	63	6F	6D	2E	64
0000070	69	73	63	6F	72	64	2E	6D	6F	64	65	6C	73	2E	64	6F
0000080	6D	61	69	6E	2E	4D	6F	64	65	6C	55	73	65	72	01	00
0000090	00	D0	5A	01	04	AC	6A	61	76	61	2E	75	74	69	6C	2E
00000A0	63	6F	6E	63	75	72	72	65	6E	74	2E	61	74	6F	6D	69
00000B0	63	2E	41	74	6F	6D	69	63	52	65	66	65	72	65	6E	63
00000C0	65	01	03	01	86	23	35	38	30	30	94	80	80	A8	BE	C5
00000D0	F7	9E	0C	01	04	01	03	01	8F	40	62	6D	61	6E	34	35
00000E0	33	31	23	35	38	30	30	01	89	62	6D	61	6E	34	35	33
00000F0	31	00	80	80	90	E0	88	8E	F9	9E	0C	01	A4	54	68	69
0000100	73	20	69	73	20	74	68	65	20	66	69	72	73	74	20	6D
0000110	65	73	73	61	67	65	20	6D	79	20	66	72	69	65	6E	64
0000120	00	01	04	01	09	00	01	01	01	00	BC	80	90	C4	EB	90
0000130	F9	9E	0C	00	01	01	01	00	01	01	01	00	00	01	00	01
0000140	05	98	6A	61	76	61	2E	75	74	69	6C	2E	4C	69	6E	6B
0000150	65	64	48	61	73	68	4D	61	70	01	01	01	83	ED	A0	BD
0000160	ED	B8	8C	01	06	AF	63	6F	6D	2E	64	69	73	63	6F	72
0000170	64	2E	6D	6F	64	65	6C	73	2E	64	6F	6D	61	69	6E	2E
0000180	4D	6F	64	65	6C	4D	65	73	73	61	67	65	52	65	61	63
0000190	74	69	6F	6E	01	04	01	07	B5	63	6F	6D	2E	64	69	73
00001A0	63	6F	72	64	2E	6D	6F	64	65	6C	73	2E	64	6F	6D	61
00001B0	69	6E	2E	4D	6F	64	65	6C	4D	65	73	73	61	67	65	52
00001C0	65	61	63	74	69	6F	6E	24	45	6D	6F	6A	69	01	00	00
00001D0	12	01	01	A1	32	30	31	38	2D	30	35	2D	30	32	54	30
00001E0	31	3A	35	39	3A	34	38	2E	38	34	39	30	30	30	2B	30
00001F0	30	3A	30	30	01	04	01	09	C0	E8	9D	E9	E3	58	00	00
0000200	00	**01**	**02**	01	00	00	01	01	01	01	01	08	B1	63	6F	6D
0000210	2E	64	69	73	63	6F	72	64	2E	6D	6F	64	65	6C	73	2E
0000220	64	6F	6D	61	69	6E	2E	4D	6F	64	65	6C	4D	65	73	73
0000230	61	67	65	41	74	74	61	63	68	6D	65	6E	74	01	01	99
0000240	4A	50	45	47	5F	32	30	31	38	30	34	32	37	5F	31	33
0000250	34	36	32	37	2E	6A	70	67	80	3F	00	01	E8	01	68	74
0000260	74	70	73	3A	2F	2F	6D	65	64	69	61	2E	64	69	73	63
0000270	6F	72	64	61	70	70	2E	6E	65	74	2F	61	74	74	61	63
0000280	68	6D	65	6E	74	73	2F	34	34	31	30	35	36	31	33	37
0000290	30	33	35	35	31	33	38	35	36	2F	34	34	31	30	35	36
00002A0	34	39	30	38	35	38	37	34	31	37	38	30	2F	4A	50	45
00002B0	47	5F	32	30	31	38	30	34	32	37	5F	31	33	34	36	32
00002C0	37	2E	6A	70	67	01	F4	9B	CD	01	01	E6	01	68	74	74

Figure 32. Hex representation of file `STORE_MESSAGES_CACHE_V17`.

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00002D0	70	73	3A	2F	2F	63	64	6E	2E	64	69	73	63	6F	72	64
00002E0	61	70	70	2E	63	6F	6D	2F	61	74	74	61	63	68	6D	65
00002F0	6E	74	73	2F	34	34	31	30	35	36	31	33	37	30	33	35
0000300	35	31	33	38	35	36	2F	34	34	31	30	35	36	34	39	30
0000310	38	35	38	37	34	31	37	38	30	2F	4A	50	45	47	5F	32
0000320	30	31	38	30	34	32	37	5F	31	33	34	36	32	37	2E	6A
0000330	70	67	A0	2F	01	03	01	00	00	BC	92	01	01	04	01	03
0000340	01	86	23	39	33	37	34	82	80	90	D4	F2	C2	F8	9E	0C
0000350	01	04	01	03	01	8F	40	72	65	64	6D	36	35	33	30	23
0000360	39	33	37	34	01	89	72	65	64	6D	36	35	33	30	00	80
0000370	80	90	E0	88	8E	F9	9E	0C	01	81	00	01	04	01	09	00
0000380	01	01	01	00	A8	80	90	C4	D9	A2	F9	9E	0C	00	01	01
0000390	01	00	01	01	01	00	00	01	00	00	01	A1	32	30	31	38
00003A0	2D	30	35	2D	30	32	54	30	32	3A	30	31	3A	30	32	2E
00003B0	30	30	31	30	30	30	2B	30	30	3A	30	30	01	04	01	09
00003C0	E0	EC	A6	E9	E3	58	00	00	00	**01**	**02**	01	00	00	01	01
00003D0	01	00	01	03	01	00	00	D0	5A	01	04	01	03	01	86	23
00003E0	35	38	30	30	94	80	80	A8	BE	C5	F7	9E	0C	01	04	01
00003F0	03	01	8F	40	62	6D	61	6E	34	35	33	31	23	35	38	30
0000400	30	01	89	62	6D	61	6E	34	35	33	31	01	09	AC	63	6F
0000410	6D	2E	64	69	73	63	6F	72	64	2E	6D	6F	64	65	6C	73
0000420	2E	64	6F	6D	61	69	6E	2E	4D	6F	64	65	6C	4D	65	73
0000430	73	61	67	65	24	43	61	6C	6C	01	01	A1	32	30	31	38
0000440	2D	30	35	2D	30	32	54	30	32	3A	30	32	3A	33	39	2E
0000450	32	34	30	30	30	30	2B	30	30	3A	30	30	01	01	**01**	**02**
0000460	01	82	80	90	D4	F2	C2	F8	9E	0C	01	94	80	80	A8	BE
0000470	C5	F7	9E	0C	80	80	90	E0	88	8E	F9	9E	0C	01	81	00
0000480	01	04	01	00	01	01	01	00	94	80	A0	A4	BD	B4	F9	9E
0000490	0C	00	01	01	01	00	01	01	01	01	01	03	01	00	00	BC
00004A0	92	01	01	04	01	03	01	86	23	39	33	37	34	82	80	90
00004B0	D4	F2	C2	F8	9E	0C	01	04	01	03	01	8F	40	72	65	64
00004C0	6D	36	35	33	30	23	39	33	37	34	01	89	72	65	64	6D
00004D0	36	35	33	30	00	01	00	00	01	A1	32	30	31	38	2D	30
00004E0	35	2D	30	32	54	30	32	3A	30	32	3A	31	34	2E	38	32
00004F0	35	30	30	30	2B	30	30	3A	30	30	01	04	01	09	E0	D1
0000500	AF	E9	E3	58	00	06	00	**01**	**02**	01	00	00	01	01	01	00
0000510	01	03	01	00	00	BC	92	01	01	04	01	03	01	86	23	39
0000520	33	37	34	82	80	90	D4	F2	C2	F8	9E	0C	01	04	01	03
0000530	01	8F	40	72	65	64	6D	36	35	33	30	23	39	33	37	34
0000540	01	89	72	65	64	6D	36	35	33	30	01	09	01	01	A1	32
0000550	30	31	38	2D	30	35	2D	30	32	54	30	32	3A	30	33	3A
0000560	31	32	2E	37	34	35	30	30	30	2B	30	30	3A	30	30	01
0000570	01	**01**	**02**	01	82	80	90	D4	F2	C2	F8	9E	0C	01	94	80
0000580	80	A8	BE	C5	F7	9E	0C	80	80	90	E0	88	8E	F9	9E	0C
0000590	01	81	00	01	04	01	00	01	01	01	00	82	80	A0	D4	FF

Figure 33. Hex representation of file `STORE_MESSAGES_CACHE_V17` (cont'd).

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00005A0	BC	F9	9E	0C	00	01	01	01	00	01	01	01	01	01	03	01
00005B0	00	00	D0	5A	01	04	01	03	01	86	23	35	38	30	30	94
00005C0	80	80	A8	BE	C5	F7	9E	0C	01	04	01	03	01	8F	40	62
00005D0	6D	61	6E	34	35	33	31	23	35	38	30	30	01	89	62	6D
00005E0	61	6E	34	35	33	31	00	01	00	00	01	A1	32	30	31	38
00005F0	2D	30	35	2D	30	32	54	30	32	3A	30	32	3A	34	39	2E
0000600	37	31	37	30	30	30	2B	30	30	3A	30	30	01	04	01	09
0000610	D0	F4	B3	E9	E3	58	00	06	00	**01**	**02**	01	00	00	01	01
0000620	01	00	01	03	01	00	00	BC	92	01	01	04	01	03	01	86
0000630	23	39	33	37	34	82	80	90	D4	F2	C2	F8	9E	0C	01	04
0000640	01	03	01	8F	40	72	65	64	6D	36	35	33	30	23	39	33
0000650	37	34	01	89	72	65	64	6D	36	35	33	30	00	80	80	90
0000660	E0	88	8E	F9	9E	0C	01	81	00	01	04	01	00	01	01	01
0000670	00	94	80	80	AC	D6	C9	F9	9E	0C	00	01	01	01	00	01
0000680	01	01	00	00	01	00	00	01	A1	32	30	31	38	2D	30	35
0000690	2D	30	32	54	30	32	3A	30	33	3A	34	31	2E	36	34	33
00006A0	30	30	30	2B	30	30	3A	30	30	01	04	01	09	90	A1	BA
00006B0	E9	E3	58	00	0C	00	**01**	**02**	01	00	00	01	01	01	00	01
00006C0	03	01	00	00	BC	92	01	01	04	01	03	01	86	23	39	33
00006D0	37	34	82	80	90	D4	F2	C2	F8	9E	0C	01	04	01	03	01
00006E0	8F	40	72	65	64	6D	36	35	33	30	23	39	33	37	34	01
00006F0	89	72	65	64	6D	36	35	33	30	00	80	80	90	E0	88	8E
0000700	F9	9E	0C	01	A8	48	65	72	65	20	69	73	20	61	20	6D
0000710	65	6E	74	69	6F	6E	20	3C	40	34	34	31	30	35	32	36
0000720	39	31	30	37	30	37	31	33	38	36	36	3E	00	01	04	01
0000730	09	00	01	01	01	00	82	80	80	EC	85	D4	F9	9E	0C	00
0000740	01	01	01	00	01	01	01	01	01	03	01	00	00	D0	5A	01
0000750	04	01	03	01	86	23	35	38	30	30	94	80	80	A8	BE	C5
0000760	F7	9E	0C	01	04	01	03	01	8F	40	62	6D	61	6E	34	35
0000770	33	31	23	35	38	30	30	01	89	62	6D	61	6E	34	35	33
0000780	31	00	01	00	00	01	A1	32	30	31	38	2D	30	35	2D	30
0000790	32	54	30	32	3A	30	34	3A	32	34	2E	31	32	33	30	30
00007A0	30	2B	30	30	3A	30	30	01	04	01	09	80	C1	BF	E9	E3
00007B0	58	00	00	00	**01**	**02**	01	00	00	01	01	01	00	01	03	01
00007C0	00	00	D0	5A	01	04	01	03	01	86	23	35	38	30	30	94
00007D0	80	80	A8	BE	C5	F7	9E	0C	01	04	01	03	01	8F	40	62
00007E0	6D	61	6E	34	35	33	31	23	35	38	30	30	01	89	62	6D
00007F0	61	6E	34	35	33	31	00	80	80	90	E0	88	8E	F9	9E	0C
0000800	01	81	00	01	04	01	00	01	01	01	00	80	80	90	98	E7
0000810	A8	8A	A1	0C	00	01	01	01	00	01	01	01	00	00	01	00
0000820	00	01	A1	32	30	31	38	2D	30	35	2D	30	33	54	31	37
0000830	3A	34	36	3A	35	37	2E	36	33	38	30	30	30	2B	30	30
0000840	3A	30	30	01	04	01	09	D0	E9	E9	F1	E4	58	00	0C	00
0000850	**01**	**02**	01	00	00	01	01	01	00	01	03	01	00	00	D0	5A
0000860	01	04	01	03	01	86	23	35	38	30	30	94	80	80	A8	BE

Figure 34. Hex representation of file `STORE_MESSAGES_CACHE_V17` (cont'd).

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000870	C5	F7	9E	0C	01	04	01	03	01	8F	40	62	6D	61	6E	34
0000880	35	33	31	23	35	38	30	30	01	89	62	6D	61	6E	34	35
0000890	33	31	00	80	80	90	E0	88	8E	F9	9E	0C	01	88	47	6F
00008A0	6F	62	65	72	73	00	01	04	01	09	00	01	01	01	00	AA
00008B0	80	90	EC	F4	AF	A5	A2	0C	00	01	01	01	00	01	01	01
00008C0	00	00	01	00	00	01	A1	32	30	31	38	2D	30	35	2D	30
00008D0	34	54	31	36	3A	32	31	3A	35	31	2E	33	38	37	30	30
00008E0	30	2B	30	30	3A	30	30	01	04	01	09	B0	B4	AD	BF	E5
00008F0	58	00	00	00	**01**	**02**	01	00	00	01	01	01	00	01	03	01
0000900	00	00	D0	5A	01	04	01	00	94	80	80	A8	BE	C5	F7	9E
0000910	0C	01	04	01	00	01	89	62	6D	61	6E	34	35	33	31	00
0000920	80	80	90	E0	88	8E	F9	9E	0C	01	8C	59	65	73	20	77
0000930	65	20	6B	6E	6F	77	00	01	04	01	09	00	01	01	01	00
0000940	80	80	80	98	85	CE	A5	A2	0C	00	01	01	01	00	01	01
0000950	01	00	01	93	34	34	38	31	36	39	32	30	31	31	39	39
0000960	32	31	38	36	38	38	01	00	00	01	A1	32	30	31	38	2D
0000970	30	35	2D	30	34	54	31	36	3A	32	33	3A	35	34	2E	37
0000980	39	30	30	30	30	2B	30	30	3A	30	30	01	04	01	09	A0
0000990	B6	BC	BF	E5	58	00	00	00								

Figure 35. Hex representation of file `STORE_MESSAGES_CACHE_V17` (cont'd).

```
com.discord.models.domain.ModelMessage
|
+-> activity (com.discord.models.domain.ModelMessage.$Activity)
+-> application (com.discord.models.domain.ModelMessage$Application)
+-> attachments (java.util.ArrayList<com.discord.models.domain.ModelMessageAttachment>)
|   |
|   +-> fileName (java.lang.String)
|   +-> height (java.lang.Integer)
|   +-> id (java.lang.String)
|   +-> proxyUrl (java.lang.String)
|   +-> size (java.lang.Long)
|   +-> url (java.lang.String)
|   +-> width (java.lang.Integer)
|
+-> author (com.discord.models.domain.ModelUser)
|   |
|   +-> avatar (java.lang.String)
|   +-> bot (java.lang.Boolean)
|   +-> discriminator (java.lang.Integer)
|   +-> discriminatorWithPadding (java.util.concurrent.atomic.AtomicReference)
|   +-> id (java.lang.Long)
|   +-> mention (java.util.concurrent.atomic.AtomicReference)
|   +-> userName (java.lang.String)
|
+-> call (com.discord.models.domain.ModelMessage$Call)
|   |
|   +-> endedTimestamp (java.lang.String)
|   +-> participants (java.util.ArrayList <java.lang.Long>)
|
+-> channelId (java.lang.Long)
+-> content (java.lang.String)
+-> editedTimestamp (java.lang.String)
+-> editedTimestampMilliseconds (java.util.concurrent.atomic.AtomicReference)
+-> embeds (java.util.ArrayList)
+-> id (java.lang.Long)
+-> mentionEveryone (java.lang.Boolean)
+-> mentionRoles (java.util.ArrayList<java.lang.Long>)
+-> mentions (java.util.ArrayList<com.discord.models.domain.ModelUser>)
+-> nonce (java.lang.String)
+-> pinned (java.lang.Boolean)
+-> reactions (java.util.LinkedHashMap)
+-> timestamp (java.lang.String)
+-> timestampMilliseconds (java.util.concurrent.atomic.AtomicReference)
+-> tts (java.lang.Boolean)
+-> type (java.lang.Integer)
+-> webhookId (java.lang.Long)
```

Figure 36. Schema from Model Message.

```
OFFSET  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00003C0 E0 EC A6 E9 E3 58 00 00 00 01 02 01 00 00 01 01
00003D0 01 00 01 03 01 00 00 D0 5A 01 04 01 03 01 86 23
00003E0 35 38 30 30 94 80 80 A8 BE C5 F7 9E 0C 01 04 01
00003F0 03 01 8F 40 62 6D 61 6E 34 35 33 31 23 35 38 30
0000400 30 01 89 62 6D 61 6E 34 35 33 31 01 09 AC 63 6F
0000410 6D 2E 64 69 73 63 6F 72 64 2E 6D 6F 64 65 6C 73
0000420 2E 64 6F 6D 61 69 6E 2E 4D 6F 64 65 6C 4D 65 73
0000430 73 61 67 65 24 43 61 6C 6C 01 01 A1 32 30 31 38
0000440 2D 30 35 2D 30 32 54 30 32 3A 30 32 3A 33 39 2E
0000450 32 34 30 30 30 30 2B 30 30 3A 30 30 01 01 01 02
0000460 01 82 80 90 D4 F2 C2 F8 9E 0C 01 94 80 80 A8 BE
0000470 C5 F7 9E 0C 80 80 90 E0 88 8E F9 9E 0C 01 81 00
0000480 01 04 01 00 01 01 01 00 94 80 A0 A4 BD B4 F9 9E
0000490 0C 00 01 01 01 00 01 01 01 01 01 03 01 00 00 BC
00004A0 92 01 01 04 01 03 01 86 23 39 33 37 34 82 80 90
00004B0 D4 F2 C2 F8 9E 0C 01 04 01 03 01 8F 40 72 65 64
00004C0 6D 36 35 33 30 23 39 33 37 34 01 89 72 65 64 6D
00004D0 36 35 33 30 00 01 00 00 01 A1 32 30 31 38 2D 30
00004E0 35 2D 30 32 54 30 32 3A 30 32 3A 31 34 2E 38 32
00004F0 35 30 30 30 2B 30 30 3A 30 30 01 04 01 09 E0 D1
0000500 AF E9 E3 58 00 06 00 01 02 01 00 00 01 01 01 00
```

Figure 37. Discord Hex ModelMessage.

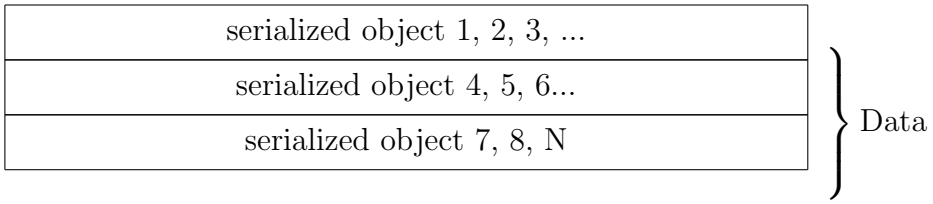

Figure 38. Format of file `STORE_MESSAGES_CACHE_V17`.

5.4 Summary

ADSS successfully identifies the format of `realtime-demo_KEY_RIDER` and file `STORE_MESSAGES_CACHE_V17` which both store evidentiary data in an unknown file format used by Android applications Uber and Discord, respectively. `realtime-demo_KEY_RIDER` stores profile information about the Uber user, such as email address, username, full name. `STORE_MESSAGES_CACHE_V17` stores the entire chat log history for the Discord user, profile details, and other evidentiary user information. Both applications use custom serializers that convert the application objects into data prior to storage into their respective files.

The next chapter concludes and covers additional research topics that can be expanded from this dissertation.

VI. Conclusion and Future Work

The work presented in this dissertation fulfills the original research objective – to develop a process that automates the analysis of a mobile application to determine how and where it stores evidentiary data in files of unknown file formats.

This paper presented a new system, Automated Data Structure Slayer (ADSS), that automates the reverse engineering of unknown file formats of Android Applications. ADSS associates application semantics with chunks of data stored in a file of unknown format; this association reveals the structure of a file, identifying the location and type of data stored in a file. ADSS is the first automated reverse engineering tool that integrates dynamic and static taint analysis, tracks tainted data through the application and native layers, and determines unknown file formats. ADSS will speed the development of tools to parse and extract data from these otherwise unknown file types.

The process automates the routine work and requires little additional effort from the examiner. The examiner identifies an application on a device and a file that may have evidentiary interest. The examiner downloads the APK and starts ADSS with the APK and the filename of interest. ADSS then performs preparsing and editing of the file to build a database to reference during the dynamic reverse engineering process. Once the application is patched, ADSS installs and executes the APK, iteratively hooks the application methods dynamically to identify the data structures associated with the file binary portions of the file. During the iteration, the examiner will foreground and background the application until the file objects are fully identified. ADSS then outputs the file format for the examiner. This process worked successsfully with Android applications Uber and Discord.

With the Uber application, ADSS correctly determined the fileformat for file `realtime-demo_KEY_RIDER`. It provided hex offsets at the start of each appli-

cation data structure in the file of interest. `realtime-demo_KEY_RIDER` stores profile information about the Uber user, such as email address, username, phone number, and the full name. ADSS revealed a series of objects (the main object of type `com.uber.model.core.generated.rtapi.models.rider.AutoValue_Rider`) separated by pre-meta data that preceded the length of the following serialized object. ADSS verified the format by accessing the same file generated on a different device.

With the Discord application, ADSS correctly determined the fileformat for file `STORE_MESSAGES_CACHE_V17`. It provides hex offset that begins the start of each application data structure of the file of interest. `STORE_MESSAGES_CACHE_V17` stores the entire chat log history for the Discord user, profile details, and other evidentiary user information. ADSS discovered a series of `com.discord.models.domain.ModelMessage` objects with sub-objects stored in the file; the schema for the main object data structure was produced. ADSS verified the format by accessing the same file generated on a different device.

6.1 Expanding the Research

With future work and support, ADSS can be expanded to encompass several different research areas. Namely multiple architectures, different devices across a wide range of infrastructures, encrypted files, malware analysis, and exploit development.

Multiple Platforms.

Since Frida supports multiple platforms: Linux, Windows, iOS, MacOS, QNX, and Android, ADSS can continue to leverage the open source project to interact with applications running on Apple, Blackberry, and personal computers. Although Smali is specific to Android, with the development of another parser for extracting

the data structures from the decompiled code produced by mobile and desktop device applications, a similar methodology could be employed to discover closed source file formats on other platforms.

Wide Range of Infrastructure.

Also, with the explosion of Internet of Things (IOT) devices, discovering how and where evidentiary data is stored on anything from smart wearables, digital assistants (e.g. Google Home, Alexa), automotive technologies, network infrastructure, and other electronic devices would be academically fruitful. Google-based IOT devices utilize Brillo and Weave, which share common code baseline with Android [96].

Many devices utilize the Linux kernel and with additional coding efforts, ADSS could be modified to interface with the program and discover formats for generated files of interest. Particular interest is with routers and switches that store data in memory, such as log-in credentials and data in transit.

Encrypted Files.

With the concern for privacy and security paramount in society following several national [97] [98] and corporate [99] [100] breaches, users are turning to mobile applications that promise to better protect their data using encryption. Snapchat, RedPhone, WhatsApp, KakaoTalk, are a few examples of applications that encrypt the message logs and user data on the device. With dynamic analysis, ADSS could be adopted to search for the encrypted key that is read or written to the filesystem during application startup. This step would precede discovering how the cleartext data is stored (if it uses a proprietary file format).

Application Security Analysis.

ADSS could be adopted to conduct an automated security assessment of mobile applications. There are millions of Android applications available for users to download from the Android Market and from third party application markets ecosystems [101]. Using ADSS to automate the analysis of these applications would be beneficial to the security community. Determining where and how these applications store data on the file system would alleviate the work required to analyze these applications during a security assessment. Checking if application store sensitive data outside of their private security folders, violate the stated permissions in the AndroidManifest.xml file, and assessing the strength of the encryption (if any) would give an application a security score and provide a third party security assessment for the application.

Appendix A. ADSS Final Hook List for Uber

This appendix provides the final hook list for the Uber Application and the targeted file: realtime-demo_KEY_RIDER.

```
var capture = ""

function toHexString (arr)
    {
        var result = "";

        var alphabet = "0123456789ABCDEF";
        var mask = 15; // 0000 1111 binary

        for (var i = 0; i < arr.length; i++ )
        {
            var idx1 = arr[ i ] & mask;
            var idx2 = ( arr[ i ] >> 4 ) & mask;

            result += alphabet[ idx2 ] + alphabet[ idx1 ] + " "
                ;
        }
        result = result.substring( 0, result.length - 1 );

        return result;
    }


if (Java.available)
{
    Java.perform(function ()
    {
        var hook_obj0 = Java.use('com.esotericsoftware.kryo.
            serializers.CompatibleFieldSerializer');
        var hook_method0 = hook_obj0.write.implementation =
            function (arg1, arg2, arg3)
        {
            if (arg3)
            {
                var arg3_type = arg3.getClass().toString();
                if (arg3_type === 'class com.uber.model.core.
                    generated.rtapi.models.rider.AutoValue_Rider'
                    )
                {
                    capture = arg2.toString()
                    console.log("\nSTART
                        CompatibleFieldSerializer  and filtering
                        on " + capture);

                    this.write(arg1, arg2, arg3)

                    console.log("\tEND CompatibleFieldSerializer
                        ");
                }

                else
```

```
44                  this.write(arg1, arg2, arg3)
45              }
46
47              else
48                  this.write(arg1, arg2, arg3)
49          }
50
51          var hook_obj1 = Java.use('com.esotericsoftware.kryo.io
                .Output');
52          var hook_method1 = hook_obj1.writeString.overload('
                java.lang.String').implementation = function (arg1)
                 //java.lang.String
53          {
54              if (capture === this.toString())
55              {
56                  var mbefore_buffer = toHexString(this.toBytes())
57
58                  var ret_val = this.writeString(arg1)
59
60                  var mafter_buffer = toHexString(this.toBytes())
61
62                  console.log("\t\t[FUNC HOOK = writeString(),
                        VALUE = " + arg1 + ", BUFFER DELTA = " +
                        mafter_buffer.slice(mbefore_buffer.length,
                        mafter_buffer.length) + "]");
63                  return ret_val;
64              }
65
66              else
67                  return this.writeString(arg1)
68          }
69
70          var hook_obj2 = Java.use('com.esotericsoftware.kryo.
                Kryo');
71          var hook_method2 = hook_obj2.writeClass.implementation
                 = function (arg1, arg2)
72          {
73              if (capture === arg1.toString())
74              {
75                  var mbefore_buffer= toHexString(arg1.toBytes())
76                  var mvalue = "null"
77                  var mtype = "null"
78                  var mafter_buffer = ""
79                  var ret_val = ""
80
81                  if (arg2)
82                  {
83                      var field_class = arg2.getClass();
84                      mvalue = arg2.toString();
85
86                      if (field_class)
87                          mtype = field_class.toString();
88                  }
89
90                  ret_val = this.writeClass(arg1, arg2);
91                  mafter_buffer = toHexString(arg1.toBytes());
92
```

```
93              console.log("\t\t\t[WRITE CLASS, TYPE = " +
                   mtype + ", VALUE = " + mvalue + ", BUFFER
                   DATA = " + mafter_buffer.slice(mbefore_buffer
                   .length, mafter_buffer.length) + "]");
94              return ret_val
95
96           }
97
98           else
99              return this.writeClass(arg1, arg2);
100       }
101
102       var hook_method3 = hook_obj1.writeVarInt.
             implementation = function (arg1, arg2)
103       {
104          if (capture === this.toString())
105          {
106             var mbefore_buffer = toHexString(this.toBytes())
107             var ret_val = this.writeVarInt(arg1, arg2);
108             var mafter_buffer = toHexString(this.toBytes())
109
110             console.log("\t\t[FUNC HOOK = writeVarInt(),
                   BUFFER DELTA = " + mafter_buffer.slice(
                   mbefore_buffer.length, mafter_buffer.length)
                   + "]");
111
112             return ret_val;
113          }
114
115          else
116             return this.writeVarInt(arg1, arg2)
117       }
118
119       var hook_obj3 = Java.use('com.esotericsoftware.kryo.
             serializers.ObjectField');
120       var hook_method4 = hook_obj3.write.implementation =
             function (arg1, arg2)
121       {
122          var mtype = "null"
123
124          if (arg2)
125          {
126             if (arg2.getClass())
127             {
128                mtype = arg2.getClass().toString()
129                if (mtype === 'class com.uber.model.core.
                      generated.rtapi.models.rider.
                      AutoValue_Rider')
130                   capture = arg1.toString()
131             }
132          }
133
134          if (capture === arg1.toString())
135          {
136             var mvalue = "null"
137             var mname = this.toString()
138             var field_type = "null"
139             var field_obj = null
```

```
140
141             if (arg2)
142                 field_obj = this.getField(arg2);
143
144
145             if (field_obj)
146             {
147                 field_type = field_obj.getClass().toString()
148                 mvalue = field_obj.toString()
149             }
150
151             var mbefore_buffer = toHexString(arg1.toBytes())
152             this.write(arg1, arg2);
153             var mafter_buffer = toHexString(arg1.toBytes());
154
155             console.log("[SERIALIZER = ObjectField, CLASS =
                    " + mtype + ", NAME = " + mname + ", TYPE = "
                     + field_type + ", VALUE = " + mvalue + ",
                    BUFFER DATA DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]\n");
156         }
157
158         else
159             this.write(arg1, arg2)
160     }
161
162     var hook_obj4 = Java.use('com.esotericsoftware.kryo.
        serializers.ObjectField$ObjectBooleanField');
163     var hook_method5 = hook_obj4.write.implementation =
        function (arg1, arg2)
164     {
165         var mtype = "null"
166
167         if (arg2)
168         {
169             if (arg2.getClass())
170             {
171                 mtype = arg2.getClass().toString()
172                 if (mtype === 'class com.uber.model.core.
                        generated.rtapi.models.rider.
                        AutoValue_Rider')
173                     capture = arg1.toString()
174             }
175         }
176
177         if (capture === arg1.toString())
178         {
179             var mvalue = "null"
180             var mname = this.toString()
181             var field_type = "null"
182             var field_obj = null
183
184             if (arg2)
185                 field_obj = this.getField(arg2);
186
187             if (field_obj)
188             {
```

```
189                 field_type = field_obj.getClass().toString()
190                 mvalue = field_obj.toString()
191             }
192
193             var mbefore_buffer = toHexString(arg1.toBytes())
194
195             this.write(arg1, arg2);
196
197             var mafter_buffer = toHexString(arg1.toBytes());
198
199             console.log("[SERIALIZER =
                    ObjectField$ObjectBooleanField, CLASS = " +
                    mtype + ", NAME = " + mname + ", TYPE = " +
                    field_type + ", VALUE = " + mvalue + ",
                    BUFFER DATA DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]\n");
200         }
201
202         else
203             this.write(arg1, arg2)
204     }
205
206     var hook_obj5 = Java.use('com.esotericsoftware.kryo.
        serializers.ObjectField$ObjectIntField');
207     var hook_method6 = hook_obj5.write.implementation =
        function (arg1, arg2)
208     {
209         var mtype = "null"
210
211         if (arg2)
212         {
213             if (arg2.getClass())
214             {
215                 mtype = arg2.getClass().toString()
216                 if (mtype === 'class com.uber.model.core.
                        generated.rtapi.models.rider.
                        AutoValue_Rider')
217                     capture = arg1.toString()
218             }
219         }
220
221         if (capture === arg1.toString())
222         {
223             var mvalue = "null"
224             var mname = this.toString()
225             var field_type = "null"
226             var field_obj = null
227
228             if (arg2)
229                 field_obj = this.getField(arg2);
230
231             if (field_obj)
232             {
233                 field_type = field_obj.getClass().toString()
234                 mvalue = field_obj.toString()
235             }
236
```

```
237                 var mbefore_buffer = toHexString(arg1.toBytes())
238
239                 this.write(arg1, arg2);
240
241                 var mafter_buffer = toHexString(arg1.toBytes());
242
243                 console.log("[SERIALIZER =
                        ObjectField$ObjectIntField, CLASS = " + mtype
                         + ", NAME = " + mname + ", TYPE = " +
                        field_type + ", VALUE = " + mvalue + ",
                        BUFFER DATA DELTA = " + mafter_buffer.slice(
                        mbefore_buffer.length, mafter_buffer.length)
                        + "]\n");
244             }
245
246             else
247                 this.write(arg1, arg2)
248         }
249
250     });
251 }
```

Appendix B. ADSS Final Hook List for Discord

This appendix provides the final hook list for the Discord Application and the targeted file: STORE_MESSAGES_CACHE_V17.

```
var capture = ""
function toHexString (arr)
{
    var result = "";
    var alphabet = "0123456789ABCDEF";
    var mask = 15; // 0000 1111 binary
    for (var i = 0; i < arr.length; i++ )
    {
        var idx1 = arr[ i ] & mask;
        var idx2 = ( arr[ i ] >> 4 ) & mask;
        result += alphabet[ idx2 ] + alphabet[ idx1 ] + " ";
    }
    result = result.substring( 0, result.length - 1 );
    return result;
}

if (Java.available)
{
    Java.perform(function ()
    {
        var hook_obj1 = Java.use('com.esotericsoftware.kryo.
            Kryo');
        var hook_method1 = hook_obj1.writeClassAndObject.
            implementation = function (arg1, arg2)
        {
            if (arg2)
            {
                var arg2_type = arg2.getClass().toString();
                if (arg2_type === 'class com.discord.models.
                    domain.ModelMessage')
                {
                    console.log("\tSTART DATA COLLECTION\n");
                    capture = arg1.toString()
                    this.writeClassAndObject(arg1, arg2);
                    console.log("\tEND DATA COLLECTION\n");
                }
                else
                    this.writeClassAndObject(arg1, arg2);
            }
            else
                this.writeClassAndObject(arg1, arg2);
        }
        var hook_method2 = hook_obj1.writeClass.implementation
            = function (arg1, arg2)
        {
            if (capture === arg1.toString())
            {
                var mbefore_buffer = toHexString(arg1.toBytes())
                var ret_val = this.writeClass(arg1, arg2);
                var mafter_buffer = toHexString(arg1.toBytes());
                console.log("[FUNC HOOK = writeClass(), BUFFER
                    DATA DELTA = " + mafter_buffer.slice(
```

```
                mbefore_buffer.length, mafter_buffer.length)
                + "]");
48              return ret_val;
49          }
50          else
51              return this.writeClass(arg1, arg2);
52      }
53      var hook_method2 = hook_obj1.writeObject.overload('com
            .esotericsoftware.kryo.io.Output', 'java.lang.
            Object', 'com.esotericsoftware.kryo.Serializer').
            implementation = function (arg1, arg2, arg3)
54      {
55          if (capture === arg1.toString())
56          {
57              var mvalue = ""
58              var mtype = "null"
59              if (arg2)
60              {
61                  mvalue = arg2.toString();
62                  if (arg2.getClass());
63                      mtype = arg2.getClass().toString();
64              }
65              var mbefore_buffer = toHexString(arg1.toBytes())
66              var ret_val = this.writeObject(arg1, arg2, arg3)
                    ;
67              var mafter_buffer = toHexString(arg1.toBytes());
68              console.log("[FUNC HOOK = writeObject(), CLASS =
                     " + mtype + ", VALUE = " + mvalue + ",
                    BUFFER DATA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]");
69              return ret_val;
70          }
71           else
72              return this.writeObject(arg1, arg2, arg3);
73      }
74      var hook_obj2 = Java.use('com.esotericsoftware.kryo.io
            .Output');
75      var hook_method3 = hook_obj2.writeVarInt.
            implementation = function (arg1, arg2)  //(int,
            boolean)
76      {
77          if (capture === this.toString())
78          {
79              var mbefore_buffer = toHexString(this.toBytes())
80              var ret_val = this.writeVarInt(arg1, arg2);
81              var mafter_buffer = toHexString(this.toBytes())
82              console.log("[FUNC HOOK = writeVarInt(), BUFFER
                    DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]");
83              return ret_val;
84          }
85          else
86              return this.writeVarInt(arg1, arg2)
87      }
88      var hook_method4 = hook_obj2.writeString.overload('
            java.lang.String').implementation = function (arg1)
```

```
        //java.lang.String
89      {
90          if (capture === this.toString())
91          {
92              var mbefore_buffer = toHexString(this.toBytes())
93              var ret_val = this.writeString(arg1)
94              var mafter_buffer = toHexString(this.toBytes())
95              console.log("[FUNC HOOK = writeString(), BUFFER
                    DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]");
96              return ret_val;
97          }
98          else
99              return this.writeString(arg1)
100     }
101     var hook_obj3 = Java.use('com.esotericsoftware.kryo.
            serializers.ObjectField');
102     var hook_method5 = hook_obj3.write.implementation =
            function (arg1, arg2)
103     {
104         if (capture === arg1.toString())
105         {
106             var mvalue = ""
107             var mname = this.toString()
108             var mtype = "null"
109             var field_type = "null"
110             var field_obj = null
111             if (arg2)
112             {
113                 mtype = arg2.getClass().toString()
114                 field_obj = this.getField(arg2);
115             }
116             if (field_obj)
117             {
118                 field_type = field_obj.getClass().toString()
119                 mvalue = field_obj.toString()
120             }
121             var mbefore_buffer = toHexString(arg1.toBytes())
122             this.write(arg1, arg2);
123             var mafter_buffer = toHexString(arg1.toBytes());
124             console.log("[SERIALIZER = ObjectField, CLASS =
                    " + mtype + ", NAME = " + mname + ", TYPE = "
                     + field_type + ", VALUE = " + mvalue + ",
                    BUFFER DATA DELTA= " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]\n");
125         }
126         else
127             this.write(arg1, arg2)
128     }
129     var hook_obj6 = Java.use('com.esotericsoftware.kryo.
            serializers.ObjectField$ObjectBooleanField');
130     var hook_method6 = hook_obj6.write.implementation =
            function (arg1, arg2)
131     {
132         if (capture === arg1.toString())
133         {
```

```
134             var mvalue = ""
135             var mname = this.toString()
136             var mtype = "null"
137             var field_type = "null"
138             var field_obj = null;
139             if (arg2)
140             {
141                 mtype = arg2.getClass().toString()
142                 field_obj = this.getField(arg2);
143             }
144             if (field_obj)
145             {
146                 field_type = field_obj.getClass().toString()
147                  mvalue = field_obj.toString()
148             }
149             var mbefore_buffer = toHexString(arg1.toBytes())
150             this.write(arg1, arg2);
151             var mafter_buffer = toHexString(arg1.toBytes());
152             console.log("[SERIALIZER =
                    ObjectField$ObjectBooleanField, CLASS = " +
                    mtype + ", NAME = " + mname + ", TYPE = " +
                    field_type + ", VALUE = " + mvalue + ",
                    BUFFER DATA DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]\n");
153         }
154         else
155             this.write(arg1, arg2)
156     }
157     var hook_obj7 = Java.use('com.esotericsoftware.kryo.
            serializers.ObjectField$ObjectIntField');
158     var hook_method7 = hook_obj7.write.implementation =
            function (arg1, arg2)
159     {
160         if (capture === arg1.toString())
161         {
162             var mvalue = ""
163             var mname = this.toString()
164             var mtype = "null"
165             var field_type = "null"
166             var field_obj = null;
167             if (arg2)
168             {
169                 mtype = arg2.getClass().toString()
170                 field_obj = this.getField(arg2);
171             }
172             if (field_obj)
173             {
174                 field_type = field_obj.getClass().toString()
175                 mvalue = field_obj.toString()
176             }
177             var mbefore_buffer = toHexString(arg1.toBytes())
178             this.write(arg1, arg2);
179             var mafter_buffer = toHexString(arg1.toBytes());
180             console.log("[SERIALIZER =
                    ObjectField$ObjectIntField, CLASS = " + mtype
                     + ", NAME = " + mname + ", TYPE = " +
                    field_type + ", VALUE = " + mvalue + ",
```

```
                    BUFFER DATA DELTA = " + mafter_buffer.slice(
                    mbefore_buffer.length, mafter_buffer.length)
                    + "]\n");
181             }
182             else
183                 this.write(arg1, arg2)
184         }
185         var hook_obj8 = Java.use('com.esotericsoftware.kryo.
                serializers.ObjectField$ObjectLongField');
186         var hook_method8 = hook_obj8.write.implementation =
                function (arg1, arg2)
187         {
188             if (capture === arg1.toString())
189             {
190                 var mname = this.toString()
191                 var mtype = "null"
192                 var field_type = "null"
193                 var field_obj = null;
194                 var mvalue = ""
195                 if (arg2)
196                 {
197                     mtype = arg2.getClass().toString()
198                     field_obj = this.getField(arg2);
199                 }
200                 if (field_obj)
201                 {
202                     field_type = field_obj.getClass().toString()
203                     mvalue = field_obj.toString()
204                 }
205                 var mbefore_buffer = toHexString(arg1.toBytes())
206                 this.write(arg1, arg2);
207                 var mafter_buffer = toHexString(arg1.toBytes());
208                 console.log("[SERIALIZER =
                        ObjectField$ObjectLongField, CLASS = " +
                        mtype + ",NAME = " + mname + ", TYPE = " +
                        field_type + ", VALUE = " + mvalue + ",
                        BUFFER DATA DELTA = " + mafter_buffer.slice(
                        mbefore_buffer.length, mafter_buffer.length)
                        + "]\n");
209             }
210             else
211                 this.write(arg1, arg2)
212         }
213     });
214 }
```

Appendix C. Parser for Uber

This appendix provides the parser used to extract information from Uber's file realtime-demo_KEY_RIDER.

```
1
2  import sys
3  from functools import partial
4
5  #filename = "/home/user/Development/PycharmProjects/
       DataCarver/GoogleNexusAFiles/realtime-
       demo_KEY_RIDER_jacobson_account"
6  filename = "/home/user/Development/PycharmProjects/
       DataCarver/GoogleNexusAFiles/realtime-
       demo_KEY_RIDER_galaxy_dill_account"
7
8  file_format = [
9       (1, 'skip'),
10      (1, 'skip'),
11      (1, 'java.lang.String', 'field1'),
12      (1, 'skip'),
13      (1, 'java.lang.String', 'field2'),
14      (1, 'skip'),
15      (1, 'java.lang.String', 'field3'),
16      (1, 'skip'),
17      (1, 'java.lang.String', 'field4'),
18      (1, 'skip'),
19      (1, 'java.lang.String', 'field5'),
20      (1, 'skip'),
21      (1, 'java.lang.String', 'field6'),
22      (1, 'skip'),
23      (1, 'java.lang.String', 'field7'),
24      (1, 'skip'),
25      (1, 'java.lang.String', 'field8'),
26      (1, 'skip'),
27      (1, 'java.lang.String', 'field9'),
28      (1, 'skip'),
29      (1, 'java.lang.String', 'field10'),
30      (1, 'skip'),
31      (1, 'java.lang.String', 'field11'),
32      (1, 'skip'),
33      (1, 'java.lang.String', 'field12'),
34      (1, 'skip'),
35      (1, 'java.lang.String', 'field13'),
36      (1, 'skip'),
37      (1, 'java.lang.String', 'field14'),
38      (1, 'skip'),
39      (1, 'java.lang.String', 'field15'),
40      (1, 'skip'),
41      (1, 'java.lang.String', 'field16'),
42      (1, 'skip'),
43      (1, 'java.lang.String', 'field17'),
44      (1, 'skip'),
45      (1, 'java.lang.String', 'field18'),
46      (1, 'skip'),
47      (1, 'java.lang.String', 'field19'),
```

```
48      (1, 'skip'),
49      (1, 'java.lang.String', 'field20'),
50      (1, 'skip'),
51      (1, 'java.lang.String', 'field21'),
52      (1, 'skip'),
53      (1, 'java.lang.String', 'field22'),
54      (1, 'skip'),
55      (1, 'java.lang.String', 'field23'),
56      (1, 'skip'),
57      (1, 'java.lang.String', 'field24'),
58      (1, 'skip'),
59      (1, 'java.lang.String', 'field25'),
60      (1, 'skip'),
61      (1, 'java.lang.String', 'field26'),
62      (1, 'skip'),
63      (1, 'java.lang.String', 'field27'),
64      (1, 'skip'),
65      (1, 'java.lang.String', 'field28'),
66      (1, 'skip'),
67      (1, 'java.lang.String', 'field29'),
68      (1, 'skip'),
69      (1, 'java.lang.String', 'field30'),
70      (1, 'skip'),
71      (1, 'java.lang.String', 'field31'),
72      (1, 'skip'),
73      (1, 'java.lang.String', 'field32'),
74      (1, 'skip'),
75      (1, 'java.lang.String', 'field33'),
76      (1, 'skip'),
77      (1, 'java.lang.String', 'field34'),
78      (1, 'skip'),
79      (1, 'java.lang.String', 'field35'),
80      (1, 'skip'),
81      (1, 'length'),
82      (1, 'java.lang.String', 'claimedMobile'),
83      (1, 'skip'),
84      (1, 'length'),
85      (1, 'lnd', 'creditBalances'),
86      (1, 'skip'),
87      (1, 'length'),
88      (1, 'java.lang.String', 'email'),
89      (1, 'skip'),
90      (1, 'length'),
91      (1, 'java.lang.String', 'firstName'),
92      (1, 'skip'),
93      (1, 'length'),
94      (1, 'java.lang.Boolean', 'hasConfirmedMobile'),
95      (1, 'skip'),
96      (1, 'length'),
97      (1, 'java.lang.String',  'hasConfirmedMobileStatus'),
98      (1, 'skip'),
99      (1, 'length'),
100     (1, 'java.lang.Boolean', 'hasNoPassword'),
101     (1, 'skip'),
102     (1, 'length'),
103     (1, 'java.lang.Boolean', 'hasToOptInSmsNotifications'),
104     (1, 'skip'),
105     (1, 'length'),
```

```
106       (1, 'java.lang.Integer', 'hashCode'),
107       (1, 'skip'),
108       (1, 'length'),
109       (1, 'java.lang.Boolean', 'hashCode$Memoized'),
110       (1, 'skip'),
111       (1, 'length'),
112       (1, 'java.lang.Boolean', 'isAdmin'),
113       (1, 'skip'),
114       (1, 'length'),
115       (1, 'java.lang.Boolean', 'isTeen'),
116       (1, 'skip'),
117       (1, 'length'),
118       (1, 'com.uber.model.core.generated.rtapi.models.
             expenseinfo.AutoValueExpenseInfo', 'lastExpenseInfo')
             ,
119       (1, 'skip'),
120       (1, 'skip'),
121       (1, 'length'),
122       (1, 'com.uber.model.core.generated.rtapi.models.rider.
             ExpenseMemo', 'lastExpenseMemo'),
123       (1, 'skip'),
124       (1, 'length'),
125       (1, 'java.lang.String', 'lastName'),
126       (1, 'skip'),
127       (1, 'length'),
128       (1, 'java.lang.Boolean', '
             lastSelectedPaymentGoogleWalletUUID'),
129       (1, 'skip'),
130       (1, 'length'),
131       (1, 'java.lang.Boolean', '
             lastSelectedPaymentProfilesGoogleWallet'),
132       (1, 'skip'),
133       (1, 'length'),
134       (1, 'com.uber.model.core.generated.rtapi.models.payment.
             AutoValuePaymentProfileUuid', '
             lastSelectedPaymentProfileUUID'),
135       (1, 'skip'),
136       (1, 'skip'),
137       (1, 'length'),
138       (1, 'com.uber.model.core.generated.rtapi.models.object.
             AutoValueMeta', 'meta'),
139       (1, 'skip'),
140       (1, 'skip'),
141       (1, 'length'),
142       (1, 'java.lang.String', 'mobileCountryIso2'),
143       (1, 'skip'),
144       (1, 'length'),
145       (1, 'java.lang.String', 'mobileDigits'),
146       (1, 'skip'),
147       (1, 'length'),
148       (1, 'com.uber.model.core.generated.rtapi.models.rider.
             AutoValueURL', 'pictureUrl'),
149       (1, 'skip'),
150       (1, 'skip'),
151       (1, 'length'),
152       (1, 'java.lang.String', 'profileType'),
153       (1, 'skip'),
154       (1, 'length'),
```

```
      (1, 'lnd', 'profiles'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.String', 'promotion'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.Double', 'rating'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'lnd', 'recentFareSplitters'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.String', 'referralCode'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'com.uber.model.core.generated.rtapi.models.rider.
         AutoValueURL', 'referralUrl'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.String', 'role'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'lnf', 'thirdPartyIdentities'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.String', 'toString'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'lnd', 'tripBalances'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'java.lang.String', 'userType'),
      (1, 'skip'),
      (1, 'length'),
      (1, 'com.uber.model.core.generated.rtapi.models.rider.
         AutoValueRiderUuid', 'uuid')]


def get_translated_hex(m_byte, m_interval):

    ans = []

    for i in range(0, 600):
        if (i & -128) == 0:
            ans.append((hex(i), i))

        else:
            ans.append((hex((i & 127) | 128), i))

    for z in ans:
        if z[0] == m_byte:

            m_interval -= 1

            if m_interval == 0:
                return z[1]
```

```
211 def is_ascii(my_char: int):
212     if (my_char > 62 and my_char < 91) or (my_char > 96 and
        my_char < 123) or (my_char == 36) or (my_char > 42
        and my_char < 59) or (my_char == 95) or (my_char ==
        35):
213         return True
214
215     return False
216
217 # needs to start on first ascii character
218 def get_ascii_str(b_index):
219
220     if str(file_data[b_index]) == '0':
221         return "00", b_index
222
223     sub_index = b_index
224     ascii_str = ""
225     while is_ascii(file_data[sub_index]):
226         ascii_str += chr(file_data[sub_index])
227         sub_index += 1
228
229     return ascii_str, sub_index
230
231 byte_counter = 0
232 chunk_size = 1
233 file_data = []
234
235
236 with open(filename, 'rb') as in_file:
237     for data in iter(partial(in_file.read, chunk_size), b'')
        :
238         file_data.append(int.from_bytes(data, byteorder='big
            '))
239
240 k = 0
241
242 while k < len(file_format):
243     object_type = file_format[k][1]
244
245     if str(object_type) == 'java.lang.String':
246         ascii_str, byte_counter = get_ascii_str(byte_counter
            )
247         print('\t' + file_format[k][2] + ' :' + ascii_str)
248         k += 1
249
250     elif str(object_type) == 'skip':
251
252         for b in range(0, file_format[k][0]):
253             print(hex(file_data[b + byte_counter]))
254
255         byte_counter += file_format[k][0]
256         k += 1
257
258     elif str(object_type) == 'length':
259
260         next_object_type = file_format[k + 1][1]
261
262         if 'com.' in next_object_type:
```

```
263
264             hex_length_bytes = hex(file_data[byte_counter])
265             int_length_bytes = int(file_data[byte_counter])
266
267             if int_length_bytes == 1:
268
269                 mini_hex = ""
270                 for p in range(0, int_length_bytes):
271                     mini_hex += ' ' + hex(file_data[p +
                        byte_counter + int(file_format[k][0])
                        ])
272
273                 mini_hex = mini_hex.strip()
274                 print('\nFIELD NAME: ' + file_format[k +
                    1][2] + '\nFIELD TYPE: ' +
                    next_object_type + '\nFIELD DATA: ' +
                    mini_hex)
275
276                 byte_counter += int_length_bytes + 1
277                 k += 2
278
279             else:
280                 interval = int(file_data[byte_counter + 1])
281
282                 converted_length_bytes = get_translated_hex(
                    hex(file_data[byte_counter]), int(
                    file_data[byte_counter + 1]))
283
284                 mini_hex = ""
285                 for p in range(0, converted_length_bytes):
286                     mini_hex += ' ' + hex(file_data[p +
                        byte_counter + int(file_format[k][0])
                        ])
287
288                 mini_hex = mini_hex.strip()
289                 print('\nFIELD NAME: ' + file_format[k +
                    1][2] + '\nFIELD TYPE: ' +
                    next_object_type + '\nFIELD DATA: ' +
                    mini_hex)
290
291                 byte_counter += converted_length_bytes + 1
292                 k += 2
293
294         else:
295             length_bytes = int(file_data[byte_counter])
296
297             mini_hex = ""
298             for p in range(0, length_bytes):
299                 mini_hex += ' ' + hex(file_data[p +
                    byte_counter + int(file_format[k][0])])
300
301             mini_hex = mini_hex.strip()
302             print('\nFIELD NAME: ' + file_format[k + 1][2] +
                '\nFIELD TYPE: ' + next_object_type + '\
                nFIELD DATA: ' + mini_hex)
303
304             byte_counter += length_bytes + 1
305             k += 2
```

Appendix D. Parser for Discord

This appendix provides the parser used to extract information from Discord's file STORE_MESSAGES_CACHE_V17.

```
1
2  import sys
3  from functools import partial
4
5  #filename = "/home/user/Development/PycharmProjects/
       DataCarver/duplicate/
       STORE_MESSAGES_CACHE_V17_galaxy_doctorpropper_account"
6  filename = "/home/user/Development/PycharmProjects/
       DataCarver/duplicate/
       STORE_MESSAGES_CACHE_V17_nexus6P_doctorpropper_account"
7
8
9  file_format = [
10     #DEPTH,
                FAMILY          TYPE                  FIELD NAME
                                    LENGTH
11     (0, 'com.discord.models.ModelMessage',
                              'item',         'custom_object',   '
           main'),
12     (1, 'com.discord.models.domain.ModelMessage.$Activity',
            'item',         'custom_object',   'activity'),
13     (1, 'com.discord.models.domain.ModelMessage$Application'
           , 'item',         'custom_object',   'application'),
14     (1, 'com.discord.models.domain.ModelMessageAttachment',
            'collection', 'custom_object',   'attachments'),
15     (2, 'java.lang.String',
                                            'item',         '
           java_primitive', 'fileName'),
16     (2, 'java.lang.Integer',
                                          'item',         '
           java_primitive', 'height',                        2),
17     (2, 'java.lang.Integer',
                                          'item',         '
           java_primitive', 'id',                            1),
18     (2, 'java.lang.String',
                                            'item',         '
           java_primitive', 'proxyUrl'),
19     (2, 'java.lang.Long',
                                              'item',         '
           java_primitive', 'size',                          5),
20     (2, 'java.lang.String',
                                            'item',         '
           java_primitive', 'url'),
21     (2, 'java.lang.Integer',
                                          'item',         '
           java_primitive', 'width',                         2),
22     (1, 'com.discord.models.domain.ModelUser',
                           'item',         'custom_object',   '
           author'),
23     (2, 'java.lang.String',
                                            'item',         '
```

```
        java_primitive', 'avatar'),
24      (2, 'java.lang.Boolean',
                                          'item',       '
        java_primitive', 'bot',                          1),
25      (2, 'java.lang.Integer',
                                          'item',       '
        java_primitive', 'discriminator',                3),
26      (2, 'java.util.concurrent.atomic.AtomicReference',
                'item',       'java_object',    '
        discriminatorWithPadding',    2),
27      (2, 'java.lang.Long',
                                            'item',       '
        java_primitive', 'id',                           9),
28      (2, 'java.util.concurrent.atomic.AtomicReference',
                'item',       'java_object',    'mention',
                              1),
29      (2, 'java.lang.String',
                                          'item',       '
        java_primitive', 'userName'),
30      (1, 'com.discord.models.domain.ModelMessage$Call',
                'item',       'custom_object',  'call'),
31      (2, 'java.lang.String',
                                          'item',       '
        java_primitive', 'endedTimestamp',               3),
32      (2, 'java.lang.Long',
                                            'collection', '
        java_primitive', 'participants',                 9),
33      (1, 'java.lang.Long',
                                            'item',       '
        java_primitive', 'channelId',                    9),
34      (1, 'java.lang.String',
                                          'item',       '
        java_primitive', 'content'),
35      (1, 'java.lang.String',
                                          'item',       '
        java_primitive', 'editedTimestamp'),
36      (1, 'java.util.concurrent.atomic.AtomicReference',
                'item',       'java_object',    '
        editedTimestampMilliseconds', 3),
37      (1, 'java.lang.Object',
                                          'collection', '
        java_object',    'embeds'),
38      (1, 'java.lang.Long',
                                            'item',       '
        java_primitive', 'id',                           9),
39      (1, 'java.lang.Boolean',
                                          'item',       '
        java_primitive', 'mentionEveryone',              1),
40      (1, 'java.lang.Long',
                                            'collection', '
        java_primitive', 'mentionRoles',                 9),
41      (1, 'com.discord.models.domain.ModelUser',
                        'collection', 'custom_object',  '
        mentions'),
42      (1, 'java.lang.String',
                                          'item',       '
        java_primitive', 'nonce'),
```

```
43      (1, 'java.lang.Boolean',
                                              'item',        '
           java_primitive', 'pinned',                          2),
44      (1, 'java.util.LinkedHashMap',
                                    'item',        'java_object
           ',     'reactions'),
45      (1, 'java.lang.String',
                                               'item',        '
           java_primitive', 'timestamp'),
46      (1, 'java.util.concurrent.atomic.AtomicReference',
                 'item',        'java_object',     '
           timestampMilliseconds',        3),
47      (1, 'java.lang.Boolean',
                                              'item',        '
           java_primitive', 'tts',                             1),
48      (1, 'java.lang.Integer',
                                              'item',        '
           java_primitive', 'type',                            1),
49      (1, 'java.lang.Long',
                                                 'item',        '
           java_primitive', 'webhookId',                       1)]
50
51
52  byte_counter = 0
53  format_index = 0
54
55  pre_meta_table = {}
56  chunk_size = 1
57  obj_count = 0
58  file_data = []
59
60
61  def get_integer(b_index, length):
62      for x in range(0, length):
63          print('int: ' + hex(file_data[b_index]) + ' at count
               : ' + str(b_index))
64          b_index += 1
65
66      return b_index
67
68
69  def get_boolean(b_index, length):
70      for x in range(0, length):
71          print('bool: ' + hex(file_data[b_index]))
72          b_index += 1
73
74      return b_index
75
76
77  def get_long(b_index, length):
78      for x in range(0, length):
79          print('long: ' + hex(file_data[b_index]) + ' at
               count: ' + str(b_index))
80          b_index += 1
81
82      return b_index
83
84
```

```
85  def get_generic(b_index, length):
86      for x in range(0, length):
87          print('generic: ' + hex(file_data[b_index]))
88          b_index += 1
89
90      return b_index
91
92
93  # needs to start on first ascii character
94  def get_ascii_name(b_index):
95      if file_data[b_index] == 1:
96          b_index += 1
97
98      ascii_str = ""
99      sub_index = b_index
100
101     while is_ascii(file_data[sub_index]):
102         ascii_str += chr(file_data[sub_index])
103         sub_index += 1
104
105     # check for stopper at the end ... anything over 160
106     if file_data[sub_index] >= 160:
107         sub_index += 1
108
109     return ascii_str, sub_index
110
111
112 def extract_java_primitive(m_byte_index, m_format_index):
113     if file_format[m_format_index][1] == 'java.lang.Integer'
        :
114         print('\nFIELD: ' + file_format[m_format_index][4] +
              '\nTYPE: ' + file_format[m_format_index][1])
115
116         if file_format[m_format_index][4] == 'discriminator'
            : # variable 2 or three bytes
117             m_byte_index = get_integer(m_byte_index, 2)  #
                get two bytes
118
119             if file_data[m_byte_index] + file_data[
                m_byte_index + 1] == 2: # if there is a
                banner banner
120                 m_byte_index = get_generic(m_byte_index, 1)
121
122         else:
123             m_byte_index = get_integer(m_byte_index,
                file_format[m_format_index][5])
124
125     elif file_format[m_format_index][1] == 'java.lang.String
        ':
126
127         if str(file_data[m_byte_index]) == '0':
128             print('\nFIELD: ' + file_format[m_format_index
                ][4] + '\nTYPE: ' + file_format[
                m_format_index][1] + '\nDATA:0x00')
129
130             m_byte_index += 1
131
```

```
132                 # check for stopper at the end ... anything over
                        160
133                 if file_data[m_byte_index] >= 160:
134                     m_byte_index += 1
135
136             else:
137                 ascii_name, m_byte_index = get_ascii_name(
                        m_byte_index)
138                 print('\nFIELD: ' + file_format[m_format_index
                        ][4] + '\nTYPE: ' + file_format[
                        m_format_index][1] + '\nDATA: ' + ascii_name)
139
140         elif file_format[m_format_index][1] == 'java.lang.
                Boolean':
141             print('\nFIELD: ' + file_format[m_format_index][4] +
                    '\nTYPE: ' + file_format[m_format_index][1])
142             m_byte_index = get_boolean(m_byte_index, file_format
                    [m_format_index][5])
143
144         elif file_format[m_format_index][1] == 'java.lang.
                generic':
145             print('\nFIELD: ' + file_format[m_format_index][4] +
                    '\nTYPE: ' + file_format[m_format_index][1])
146
147             m_byte_index = get_generic(m_byte_index, file_format
                    [m_format_index][5])
148
149         elif file_format[m_format_index][1] == 'java.lang.Long':
150             print('\nFIELD: ' + file_format[m_format_index][4] +
                    '\nTYPE: ' + file_format[m_format_index][1])
151             m_byte_index = get_long(m_byte_index, file_format[
                    m_format_index][5])
152
153         elif file_format[m_format_index][1] == 'skip':
154             m_byte_index = get_generic(m_byte_index, file_format
                    [m_format_index][5])
155
156         return m_byte_index, m_format_index + 1
157
158
159 def is_ascii(my_char: int):
160     if (my_char > 63 and my_char < 91) or (my_char > 96 and
            my_char < 123) or (my_char == 36) or (my_char > 42
            and my_char < 59) or (my_char == 95) or (my_char ==
            35) or (my_char == 40) or (my_char == 32):
161         return True
162
163     return False
164
165
166 def get_object_key(b_index):
167
168     key = str(file_data[b_index])
169
170     b_index += 1
171
172     # make sure next isn't banner 01
173     if file_data[b_index] != 1:
```

```
174             ascii_str, b_index = get_ascii_name(b_index)
175             pre_meta_table[key] = ascii_str
176             return ascii_str, b_index, key
177
178         else:
179             return "", b_index, key
180
181
182     def extract_collection(m_byte_index, m_format_index):
183         m_byte_index += 3
184
185         # check if object has anything
186         if str(file_data[m_byte_index]) == '1':
187             m_byte_index += 1
188
189             if file_format[m_format_index][3] == 'java_primitive
                    ':
190                 print('\tcollection of primitives')
191                 m_byte_index, m_format_index =
                        extract_java_primitive(m_byte_index,
                        m_format_index)
192
193             elif file_format[m_format_index][3] == '
                    custom_object':
194                 print('\tcollection of custom objects')
195                 m_byte_index, m_format_index =
                        extract_custom_object(m_byte_index,
                        m_format_index)
196
197             elif file_format[m_format_index][3] == 'java_object'
                    :
198                 print('\tcollection of java objects')
199                 m_byte_index, m_format_index =
                        extract_java_object(m_byte_index,
                        m_format_index)
200
201             return m_byte_index, m_format_index + 1
202
203         else:
204             print('\tcollection has 0x00 elements')
205             # needs to skip any sub formats
206             start_depth = file_format[m_format_index][0]
207             next_depth = file_format[m_format_index + 1][0]
208
209             if start_depth < next_depth:
210                 while start_depth < next_depth:
211                     m_format_index += 1
212                     next_depth = file_format[m_format_index][0]
213
214                 return m_byte_index + 1, m_format_index
215
216             else:
217                 return m_byte_index + 1, m_format_index + 1
218
219
220     def extract_java_object(b_index, m_file_index):
221
```

```
222     # if an object field with data, expect '01 <2 byte key> 01'
223     if str(file_data[b_index]) == '1':
224         b_index += 1  # banner1
225
226         ascii_str, b_index, key = get_object_key(b_index)
227
228         if str(file_data[b_index]) == '1':
229             print('\tthere is data in this java object')
230             b_index += 1
231
232             if file_format[m_file_index][4] == '
                    timestampMilliseconds':
233                 b_index = get_generic(b_index, 7)  # 7 two
                        byte padding
234
235             else:
236                 if str(file_data[b_index]) != '0':
237                     b_index = get_generic(b_index, 1)  # 1
                            two byte padding
238
239                     if file_data[b_index] != 0:
240                         ascii_str, b_index = get_ascii_name(
                                b_index)
241                         print('\nFIELD: ' + file_format[
                                m_file_index][4] + '\nTYPE: java.
                                lang.String\nDATA: ' + ascii_str)
242
243                     else:
244                         print('\nFIELD: ' + file_format[
                                m_file_index][4] + '\nTYPE: java.
                                lang.String\nDATA: 0x00')
245                         b_index += 1
246
247                 else:
248                     print('\nFIELD: ' + file_format[
                            m_file_index][4] + '\nTYPE: java.lang
                            .String\nDATA: 0x00')
249                     b_index += 1
250
251             return b_index, m_file_index + 1
252
253         elif is_ascii(file_data[b_index]) is False:
254             print('\tno data in java object')
255
256     else:
257         print('\tno data in java object')
258         return b_index + 1, m_file_index + 1
259
260 # start at 01 0X
261 def extract_custom_object(m_byte_index, m_format_index):
262
263     start_depth = file_format[m_format_index][0]
264
265     # get banner
266     if str(file_data[m_byte_index]) == '1':
267         print('\nAAAAFIELD: ' + file_format[m_format_index
                ][4] + '\nTYPE: ' + file_format[m_format_index
```

```
            ][1])
268         m_byte_index += 1
269
270         ascii_name, m_byte_index, key = get_object_key(
                m_byte_index)
271
272         if str(file_data[m_byte_index]) == '1':
273             print('\tthere is data in object')
274
275             next_depth = file_format[m_format_index + 1][0]
276             if start_depth < next_depth:
277                 return extract_custom_object_helper(
                        m_byte_index + 1, m_format_index + 1,
                        start_depth)
278
279     else:
280         print('\nBBBBBBFIELD: ' + file_format[m_format_index
                ][4] + '\nTYPE: ' + file_format[m_format_index
                ][1] + '\nDATA:0x00' + ' HEX: ' + hex(file_data[
                m_byte_index]) + ' at byte count: ' + str(
                m_byte_index))
281
282         m_format_index += 1
283         next_depth = file_format[m_format_index][0]
284         while start_depth < next_depth:
285             print('skipping sub format ' + file_format[
                    m_format_index][1])
286             m_format_index += 1
287             next_depth = file_format[m_format_index][0]
288
289         print('oh snap2')
290         return extract_custom_object_helper(m_byte_index +
                1, m_format_index, start_depth)
291
292
293 def extract_custom_object_helper(m_byte_index,
        m_format_index, start_depth):
294
295     if m_format_index < len(file_format):
296         current_depth = file_format[m_format_index][0]
297
298         if start_depth < current_depth:
299
300             # if an object field with no data, expect '01 <2
                    byte key> 00'
301
302             if file_format[m_format_index][2] == 'item':
303
304                 if file_format[m_format_index][3] == '
                        java_primitive':
305                     m_byte_index, m_format_index =
                            extract_java_primitive(m_byte_index,
                            m_format_index)
306
307                 elif file_format[m_format_index][3] == '
                        custom_object':
308                     m_byte_index, m_format_index =
                            extract_custom_object(m_byte_index,
```

```
                            m_format_index)
309
310                     elif file_format[m_format_index][3] == '
                            java_object':
311                         m_byte_index, m_format_index =
                                extract_java_object(m_byte_index,
                                m_format_index)
312
313                 elif file_format[m_format_index][2] == '
                        collection':
314                     print('\nFIELD: ' + file_format[
                            m_format_index][4] + '\nTYPE: collection
                            of ' + file_format[m_format_index][1])
315                     m_byte_index, m_format_index =
                            extract_collection(m_byte_index,
                            m_format_index)
316
317                 return extract_custom_object_helper(m_byte_index
                        , m_format_index, start_depth)
318
319             else:
320                 return m_byte_index, m_format_index
321
322         else:
323             return m_byte_index, m_format_index
324
325
326 with open(filename, 'rb') as in_file:
327     for data in iter(partial(in_file.read, chunk_size), b'')
            :
328         file_data.append(int.from_bytes(data, byteorder='big
                ''))
329
330 message_count = 0
331 while byte_counter < len(file_data):
332     # look for pattern 01 0X ascii_string
333     # first byte should be 01
334     if str(file_data[byte_counter]) == '1':
335
336         # looking for key 01 - 09
337         if file_data[byte_counter + 1] >= 0 and file_data[
                byte_counter + 1] < 9:
338
339             save_byte_index = byte_counter
340             ascii_name, byte_counter, key = get_object_key(
                    byte_counter + 1)
341
342             if 'com.discord.models.domain.ModelMessag' in
                    ascii_name:
343                 print("\n-----------------FOUND MESSAGE " +
                        str(message_count))
344                 message_count += 1
345                 byte_counter, m_format_index =
                        extract_custom_object(save_byte_index, 0)
346
347             elif 'com.discord.models.domain.ModelMessag' in
                    pre_meta_table[key]:
```

```
348                     print("\n------------------FOUND MESSAGE " +
                            str(message_count))
349                     message_count += 1
350                     byte_counter, m_format_index =
                           extract_custom_object(save_byte_index, 0)
351
352                 else:
353                     byte_counter += 1
354
355             else:
356                 byte_counter += 1
357         else:
358             byte_counter += 1
359
360 for k in sorted(pre_meta_table.keys()):
361     print(k + ' = ' + pre_meta_table[k])
```

Bibliography

1. W. R. Stevens and S. A. Rago, *Advanced Programming in the Unix Environment (3^{rd} edition)*. New York: Addison Wesley, 2014.

2. J. Levin, "Dalvik and ART," *AndDevCon*, 2015. [Online]. Available: http://newandroidbook.com/files/ArtOfDalvik.pdf

3. A. Frumusanu, "A Closer Look at Android RunTime (ART) in Android L," *Anandtech*, 2014. [Online]. Available: http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/

4. T. L. Strazzere, J. Sawyer, and C. Fenton, "Offensive and Defensive Android Reverse Engineering," *DefCon23*, 2015. [Online]. Available: https://github.com/rednaga/training/tree/master/DEFCON23

5. U. S. Laboratories, *System V Application Binary Interface*, Xinuos Inc., June 2013. [Online]. Available: http://www.sco.com/developers/gabi/latest/contents.html

6. M. Stevanovic, *Advanced C and C++ Compiling*. New York: Apress, 2014.

7. (2018) Compiler, Assembler, Linker and Loader: A Brief Story. [Online]. Available: http://www.tenouk.com/ModuleW.html

8. R. Krishnakumar, "Experiments with the Linux Kernel: Process Segments," *Linux Gazette*, 2005. [Online]. Available: http://linuxgazette.net/112/krishnakumar.html

9. L. Torvalds and M. Others, *The Linux Kernel*. Wokingham, Berkshire United Kingdom: The Linux Documentation Project, 1999.

10. C. 02557590, *ARM ® Cortex ® -A Series Programmer's Guide for ARMv8-A ARM Cortex-A Series Programmer's Guide for ARMv8-A*, ARM Inc., May 2015. [Online]. Available: https://static.docs.arm.com/den0024/a/DEN0024.pdf

11. K. Pankaj, "Java Heap Space vs Stack - Memory Allocation in Java," *JournalDev*, 2014. [Online]. Available: https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

12. B. Gruver. (2015) TypesMethodsAndFields. [Online]. Available: https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields

13. P. Shankdhar, "22 Popular Computer Forensics Tools," *Infosec Institute*, 2018. [Online]. Available: http://resources.infosecinstitute.com/computer-forensics-tools

14. S. Hill, "Android vs. iOS: In-Depth Comparison of the Best Smartphone Platforms," *DigitalTrends*, 2018. [Online]. Available: https://www.digitaltrends.com/mobile/android-vs-ios/

15. J. Hamill, "How terrorists use encrypted messaging apps to plot, recruit and attack," *New York Post*, 2017. [Online]. Available: https://nypost.com/2017/03/28/how-terrorists-use-encrypted-messaging-apps-to-plot-recruit-and-attack/

16. A. M. de Paula, "Security Aspects and Future Trends of Social Networks," *Proceedings of the Fourth International Conference of Forensic Computer Science*, pp. 66–77, January 2009.

17. K. Rathi, U. Karabiyik, T. Aderibigbe, and H. Chi, "Forensic analysis of encrypted instant messaging applications on Android," *6th International Symposium on Digital Forensic and Security*, pp. 1–6, 2018.

18. N. Freischlad, "Top 5 secure messaging apps for all your private chats," *Tech In Asia*, 2016. [Online]. Available: https://www.techinasia.com/10-best-secure-messaging-apps

19. M. Martinez, "Colorado sexting scandal: High school faces felony investigation," *CNN*, 2015. [Online]. Available: https://www.cnn.com/2015/11/07/us/colorado-sexting-scandal-canon-city

20. X. Zhang, I. Baggili, and F. Breitinger, "Breaking into the vault: Privacy, security and forensic analysis of Android vault applications," *Computers and Security*, vol. 70, September 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404817301529

21. M. Pincus, J. Waldron, E. Schiermeyer, M. Luxton, and S. Schoettler. (2007) Free Games by Zynga. [Online]. Available: https://www.zynga.com/

22. S. Rad, J. Badeen, J. Mateen, J. Munoz, D. Moorjani, and W. Wolfe. (2014) Tinder. [Online]. Available: https://tinder.com

23. W. Mobile. (2017) Free Community-based GPS, Maps & Traffic Navigation App. [Online]. Available: https://www.waze.com/

24. "Mobile messenger apps," *Statistia*, 2016. [Online]. Available: https://www.statista.com/study/15257/mobile-messenger-apps-statista-dossier/

25. R. Baldwin, "What you need to know about Apple's fight with the FBI," *Engadget*, 2016. [Online]. Available: https://www.engadget.com/2016/02/18/fbi-apple-iphone-explainer/

26. N. Ingraham, "Senator confirms FBI paid $900,000 to unlock San Bernardino iPhone," *Engadget*, 2017. [Online]. Available: https://www.engadget.com/2017/05/08/fbi-paid-900000-to-unlock-san-bernardino-iphone/

27. E. Nakashima, "FBI paid professional hackers one-time fee to crack San Bernardino iPhone," *Washington Post*, 2016. [Online]. Available: https://www.washingtonpost.com/world/national-security/fbi-paid-professional-hackers-one-time-fee-to-crack-san-bernardino-iphone/2016/04/12/5397814a-00de-11e6-9d36-33d198ea26c5_story.html?noredirect=on&utm_term=.caf3e1abddce

28. T. Fox-Brewster, "The Feds Can Now (Probably) Unlock Every iPhone Model In Existence," *Forbes*, 2018. [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2018/02/26/government-can-access-any-apple-iphone-cellebrite/

29. S. Morgan, "U.S. Government's Multi-Million Dollar Mobile Forensics Shopping Spree Revealed," *Forbes*, 2016. [Online]. Available: https://www.forbes.com/sites/stevemorgan/2016/04/06/u-s-governments-multi-million-dollar-mobile-forensics-shopping-spree-revealed/#721b79261b0d

30. S. E. Goodison, R. C. Davis, and B. A. Jackson, "Digital Evidence and the U.S. Criminal Justice System: Identifying Technology and Other Needs to More Effectively Acquire and Utilize Digital Evidence," Rand Corporation, Tech. Rep. RR-890-NIJ, 2015. [Online]. Available: https://www.ncjrs.gov/pdffiles1/nij/grants/248770.pdf

31. C. Sadeghi, "Cellphone tracking leads to crucial evidence," *KXAN*, 2015. [Online]. Available: https://www.kxan.com/news/crime/cell-phone-tracking-leads-to-crucial-evidence_20180316010227311/1049483190

32. C. Peak, "Phone Sleuthing Clears Convicted Man," *New Haven Independent*, 2018. [Online]. Available: https://www.newhavenindependent.org/index.php/archives/entry/vernon_horn_dixwell_deli_murder/

33. K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*. Sebastopol, California: O'Reilly Media, 2013.

34. D. Beres, "Sorry, Fanboys: Android Still More Popular Than iOS In U.S." *The Huffington Post*, 2015. [Online]. Available: https://www.huffingtonpost.com/entry/android-more-popular-than-ios_us_5678203be4b06fa6887de2e7

35. R. Amadeo, "Google's iron grip on Android: Controlling open source by any means necessary," *Ars Technica*, 2013. [Online]. Available: http://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/

36. C. Metz, "Android and Linux Reunite After Two-Year Separation," *Wired*, 2012. [Online]. Available: http://www.wired.com/2012/03/android-linux/

37. P. Brady. (2008) Android Anatomy and Physiology Agenda. [Online]. Available: https://sites.google.com/site/io/anatomy--physiology-of-an-android

38. W. E. Shotts, *The Linux command line - A complete introduction.* San Francisco, California: No Starch Press, 2012.

39. D. Regalado, *Gray Hat Hacking : The Ethical Hacker's Handbook (4^{th} edition).* New York: McGraw-Hill Education, 2015.

40. S. Mahajan. (2017) How an Android application is executed on Dalvik Virtual Machine. [Online]. Available: https://stackoverflow.com/questions/13577733

41. *Getting Started with Auto*, Google Inc., April 2018. [Online]. Available: http://developer.android.com/training/auto/start/index.html

42. N. Sharma, "Android Architecture Guides for beginners," *Eureka*, 2013. [Online]. Available: https://www.edureka.co/blog/beginners-guide-android-architecture/

43. D. Kohler. (2010) Scripting Layer for Android. [Online]. Available: https://github.com/damonkohler/sl4a

44. M. Kosmiskas, "Understanding the Android bytecode," *Tech Thoughts*, 2015. [Online]. Available: http://mariokmk.github.io/programming/2015/03/06/learning-android-bytecode.html

45. P. Vara and F. H. (2016) kernel stack and user space stack. [Online]. Available: http://stackoverflow.com/questions/12911841/kernel-stack-and-user-space-stack

46. B. E. Andreasson and E. Andreasson, "JVM performance optimization, Part 4: C4 garbage collection for low-latency Java applications," *Java World*, 2012. [Online]. Available: https://www.javaworld.com/article/2078661/java-concurrency/jvm-performance-optimization--part-4--c4-garbage-collection-for-low-latency-java-ap.html

47. "Memory Management in the Java HotSpot TM Virtual Machine," Sun Microsystems, Inc., Tech. Rep. 150215, 2006. [Online]. Available: http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

48. A. Azfar, K. K. R. Choo, and L. Liu, "Forensic taxonomy of android productivity apps," *Multimedia Tools and Applications*, vol. 76, no. 3, pp. 3313–3341, February 2017.

49. M. Widenius and D. Axmark, *Mysql Reference Manual.* Sebastapol, California: O'Reilly Media, 2002.

50. (2017) Kryo Readme. [Online]. Available: https://github.com/EsotericSoftware/kryo/blob/master/README.md

51. (2008) SQLCipher. [Online]. Available: https://www.zetetic.net/sqlcipher/

52. B. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. Marsh, "Industrial perspective on static analysis," *Software Engineering Journal*, vol. 10, no. 2, pp. 69–75, March 1995.

53. C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of the 10th Association for Computing Machinery Conference on Computer and Communications Security*, pp. 290–299, October 2003.

54. *Android Debug Bridge*, Google Inc., January 2013. [Online]. Available: https://developer.android.com/studio/command-line/adb.html

55. J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android Hacker's Handbook (1st edition)*. Indianapolis, Indiana: Wiley, 2014.

56. J.-L. Gailly and P. Deutsch, *ZLIB Compressed Data Format Specification version 3.3*, Network Working Group, May 1996. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1950.txt

57. R. Winiewski and C. Tumbleson. (2010) A tool for reverse engineering Android apk files. [Online]. Available: http://ibotpeaches.github.io/Apktool/

58. I. Guilfanov. (2016) IDAPro. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

59. B. Gruver. (2017) Smali. [Online]. Available: https://github.com/JesusFreke/smali

60. S. Margaritelli, "Dynamically Inject a Shared Library Into a Running Process on Android/ARM," *evilsocket*, 2015. [Online]. Available: https://www.evilsocket.net/2015/05/01/dynamically-inject-a-shared-library-into-a-running-process-on-androidarm/

61. C. Mulliner, "Android DDI: Dynamic Dalvik Instrumentation of Android Applications and Framework," *Hack in the Box 2013*, 2013. [Online]. Available: https://www.mulliner.org/android/feed/mulliner_ddi_30c3.pdf

62. D. Tomescu, "Mobile penetration testing on Android using Drozer," *Security Cafe*, 2015. [Online]. Available: https://securitycafe.ro/2015/07/08/mobile-penetration-testing-using-drozer/

63. MWR Labs, *Drozer User Guide*, MWR InfoSecurity, March 2015. [Online]. Available: https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf

64. A. Cozzette, "Intent Spoofing on Android," *Palomino Labs Blog*, 2013. [Online]. Available: http://blog.palominolabs.com/2013/05/13/android-security/

65. J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," *Association for Computing Machinery Special Interest Group on Programming Languages Notices*, vol. 49, no. 10, pp. 83–101, October 2014.

66. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the Association for Computing Machinery*, vol. 57, no. 3, pp. 99–106, March 2014.

67. W. Chiwei and W. Shiuhpyng, "DROIT : Dynamic Alternation of Dual-Level Tainting for Malware Analysis," *Journal of Information Science and Engineering*, vol. 31, no. 1, January 2015. [Online]. Available: http://jise.iis.sinica.edu.tw/JISESearch/pages/View/PaperView.jsf?keyId=1_6

68. V. Costamagna and C. Zheng, "ARTDroid: A virtual-method hooking framework on android ART runtime," *Central Europe Workshop Proceedings*, vol. 1575, April 2016. [Online]. Available: http://ceur-ws.org/Vol-1575/paper_10.pdf

69. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86A Platform for Analyzing x86 Executables," *Proceedings of the 14th International Conference on Compiler Construction*, pp. 250–254, April 2005.

70. G. Balakrishnan and T. Reps, "Improved memory-access analysis for x86 executables," *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, vol. 3548, pp. 16–35, April 2008.

71. G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," *Proceedings of the 26th Association for Computing Machinery Special Interest Group on Algorithms and Computation Theory Symposium on Principles of Programming Languages*, pp. 119–132, January 1999.

72. J. Lim, T. Reps, and B. Liblit, "Extracting output formats from executables," *13th Working Conference on Reverse Engineering*, pp. 167–176, October 2006.

73. E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," *Adaptable and Extensible Component Systems*, vol. 30, 2002. [Online]. Available: http://asm.ow2.org/current/asm-eng.pdf

74. F. Bellard. (2017) Quick Emulator. [Online]. Available: https://www.qemu.org/

75. W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," *Proceedings of the 15th Association for Computing Machinery Conference on Computer and Communications Security*, pp. 391–402, October 2008.

76. S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pp. 154–163, June 2006.

77. A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," *Network and Distributed System Security Symposium*, February 2011. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/slow.pdf

78. Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," *Proceedings of the 11th Annual Information Security Symposium*, pp. 5:1–5:1, March 2010.

79. A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 255–266, December 2008.

80. P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pp. 255–265, July 2006.

81. R. Zhang, S. Huang, Z. Qi, and H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," *Computers and Mathematics with Applications*, vol. 63, pp. 469–480, January 2012.

82. J. Zhao, J. Qi, L. Zhou, and B. Cui, "Dynamic taint tracking of web application based on static code analysis," *10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 96–101, July 2016.

83. W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, and R. Tober, "Integration of static and dynamic code analysis for understanding legacy source code," *IEEE International Conference on Software Maintenance and Evolution*, pp. 543–552, October 2016.

84. V. Dorneanu. (2015) Static Code Analysis for Smali files. [Online]. Available: https://github.com/dorneanu/smalisca

85. *Dalvik bytecode*, Google Inc., February 2018. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

86. O. A. Ravnå. (2017) A world-class dynamic instrumentation framework - Inject JavaScript to explore native apps on Windows, Mac, Linux, iOS and Android. [Online]. Available: http://www.frida.re/

87. I. Darwin, *file(1): determine file type - Linux man page*, BSD, May 2008. [Online]. Available: https://linux.die.net/man/1/file

88. *UI/Application Exerciser Monkey*, Google Inc., January 2018. [Online]. Available: https://developer.android.com/studio/test/monkey

89. Chainfire and Coding Code Mobile Technology LLC. (2017) SuperSU. [Online]. Available: http://www.supersu.com/

90. (2016) TeamWin - TWRP. [Online]. Available: https://twrp.me/

91. C. Lefebvre, "Linux mint 18.2 sonya kde released," *The Linux Mint Blog*, July 2017. [Online]. Available: {https://blog.linuxmint.com/?p=3292}

92. T. Kalanick and G. Camp. (2018) Uber - Get a Ride Near You - Earn Money by Driving. [Online]. Available: https://www.uber.com/

93. N. Walters, "How Much Is Uber Worth Right Now?" *The Motley Fool*, 2017. [Online]. Available: https://www.fool.com/investing/2017/12/12/how-much-is-uber-worth-right-now.aspx

94. (2017) Discord - Free Voice and Text Chat for Gamers. [Online]. Available: https://discordapp.com/

95. J. Constine, "Gamer chat tool Discord secretly raised $50M as insiders cashed out," *Tech Crunch*, 2017. [Online]. Available: https://techcrunch.com/2017/06/07/discord/

96. A. Earls, "Google takes on IoT with Brillo and Weave," *IoT Agenda*, 2016. [Online]. Available: https://internetofthingsagenda.techtarget.com/feature/Google-takes-on-IoT-with-Brillo-and-Weave

97. G. Miller and E. Nakashima, "WikiLeaks says it has obtained trove of CIA hacking tools," *Washington Post*, 2017. [Online]. Available: https://www.washingtonpost.com/world/national-security/wikileaks-says-it-has-obtained-trove-of-cia-hacking-tools/2017/03/07/c8c50c5c-0345-11e7-b1e9-a05d3c21f7cf_story.html?utm_term=.caf0b837b64c

98. B. Gellman, A. Blake, and G. Miller, "Edward Snowden comes forward as source of NSA leaks," *Washington Post*, 2013. [Online]. Available: https://www.washingtonpost.com/politics/intelligence-leaders-push-back-on-leakers-media/2013/06/09/fff80160-d122-11e2-a73e-826d299ff459_story.html?utm_term=.9abe157323e4

99. M. Kumar, "Facebook admits public data of its 2.2 billion users has been compromised," *The Hacker News*, 2018. [Online]. Available: https://thehackernews.com/2018/04/facebook-data-privacy.html

100. T. Armerding, "The 17 biggest data breaches of the 21st century — CSO Online," *CSO Online*, 2018. [Online]. Available: https://www.csoonline.com/article/2130877/data-breach/the-biggest-data-breaches-of-the-21st-century.html

101. C. Mulliner, "To stay safer on Android, stick with Google Play," *The Parallax*, 2018. [Online]. Available: https://www.the-parallax.com/2018/01/25/safer-android-google-play/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
13–09–2018	Doctoral Dissertation	Sept 2015 — Sep 2018

4. TITLE AND SUBTITLE

Automating Mobile Device File Format Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Dill, Richard, Major, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Department of Electrical and Computer Engineering (AFIT/ENG)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/ENG/DS/18-S-008

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Department of Electrical and Computer Engineering
2950 Hobson Way
WPAFB OH 45433-7765

10. SPONSOR/MONITOR'S ACRONYM(S)

AFIT/ENG

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Forensic tools assist examiners in extracting evidence from application files from mobile devices. If the file format for the file of interest is known, this process is straightforward, otherwise it requires the examiner to manually reverse engineer the data structures resident in the file. This research presents the Automated Data Structure Slayer (ADSS), which automates the process to reverse engineer unknown file formats of Android applications. After statically parsing and preparing an application, ADSS dynamically runs it, injecting hooks at selected methods to uncover the data structures used to store and process data before writing to media. The resultant association between application semantics and bytes in a file reveal the structure and file format. ADSS has been successfully evaluated against Uber and Discord, both popular Android applications, and reveals the format used by the respective proprietary application files stored on the filesystem.

15. SUBJECT TERMS

Android, malware, forensics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Gilbert L. Peterson, AFIT/ENG
U	U	U	UU	136	19b. TELEPHONE NUMBER *(include area code)* 312-785-6565, x4281; gilbert.peterson@afit.edu