Mobile Data Analysis using Dynamic Binary Instrumentation and Static Analysis

THESIS

Christopher Dukarm, 2d Lt, USAF

AFIT-ENG-MS-20-M-016

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-016

Mobile Application Data Analysis using Dynamic Binary Instrumentation and Static Analysis

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

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 Master of Science in Cyber Operations

Christopher Dukarm, B.S.C.S.

2d Lt, USAF

March 26, 2020

AFIT-ENG-MS-20-M-016

Mobile Application Data Analysis using Dynamic Binary Instrumentation and Static Analysis

THESIS

Christopher Dukarm, B.S.C.S.
2d Lt, USAF

Committee Membership:

Maj Richard Dill, Ph.D
Chair

Lt Col Patrick J. Sweeney, Ph.D
Member

Dr. Timothy H. Lacey, Ph.D
Member

AFIT-ENG-MS-20-M-016

Abstract

Mobile classified data leakage poses a threat to the Department of Defense programs and missions. Security experts must know the format of application data, in order to properly classify mobile applications. This research presents the Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) methodology to identify stored data formats. DBIMAFIA uses Dynamic Binary Instrumentation and static analysis to uncover the structure of mobile application data and validate the results with traditional reverse engineering methods. DBIMAFIA is applied to fifteen popular Android applications and revealed the format of stored data. Notably, user personally identifiable information leakage is identified in the Hago Games application. The application's messaging service exposes the full name, birthday, and city of any user of the Hago Games application. These findings on how Hago Games uses ObjectBox library to store data in custom file formats can be applied more broadly to any mobile, IoT, or SCADA device or application using the ObjectBox library. Furthermore, the DBIMAFIA methodology can be more broadly defined to identify stored data within any Android application.

Acknowledgements

I thank God for giving me the wisdom and strength required to persevere through this process. Writing this thesis has been a humbling experience and has required me to rely on others for support and knowledge. I'd like to thank my advisor, Maj Richard Dill, for his patience with my endless questions and pleas for help. I'd like to thank Brandon Kamaka, Aaron Pendleton, Marvin Newlin and all of the others in the CCR who gave me sage advice and kept me sane while writing. Lastly, I'd like to thank my committee members, Dr. Timothy Lacey, and Lt Col Patrick Sweeney, for challenging my research and writing.

Christopher Dukarm

Table of Contents

Page

Page

Page

List of Figures

List of Tables

Mobile Application Data Analysis using Dynamic Binary Instrumentation and Static Analysis

I. Introduction

In May of 2019, President Trump signed Executive Order 13873 [8] stating that foreign adversaries were creating and exploiting vulnerabilities in communications equipment to commit economic and industrial espionage against Americans. This recognized that communications equipment store "vast amounts of sensitive information, facilitate the digital economy, and support critical infrastructure and vital emergency services" [8]. In August 2019, the Android operating system supported 76.23 percent of mobile devices worldwide [9]. The federal government has recognized Android's popularity and has developed hundreds of Android applications to allow Americans access to federal agency online resources and programs [10]. Cleared personnel use a modified Android operating system on tablets that host classified messages and live stream intelligence data from the Pentagon [11][12]. In response, cyber security experts must understand how these devices store and process user data to develop appropriate security procedures.

More private information is stored in mobile and Internet of Things (IoT) devices; ubiquitousness of the internet has left user data vulnerable, and even private companies heavily rely on connected devices to access private corporate data. Manufacturers sacrifice security for convenience. The recent surge of IoT devices has exacerbated the problem; more sensors and connected devices means a greater volume of data. In order to understand the increased level of risk, cyber professionals must know what and how data is stored, processes, and transmitted, to defend the valuable informa-

tion. Next, this chapter details the motivation, research objectives, and methodology of this thesis.

1.1 Motivation

This research provides a methodology that could potentially aid in the identification of classified data leakage and insecure data storage methods of applications used by US government employees and those applications developed by various federal agencies.

In 2014, Yahoo fell victim to a series of cyber attacks, exposing names, birth dates, phone numbers, and physical addresses of over 500 million users [13]. That same year, over 100 million Marriott customers had there contact, passport, and credit card information stolen from Marriott's servers [13]. From 2014 to 2018 Ebay, Equifax, Target, Uber, and Chase bank all had similar breaches leaking 100s of millions of valuable personal and business data to attackers [13]. These attacks, although not exclusively IoT or mobile targets, spurred governments to strengthen existing data privacy laws and pressure companies to better protect customer data [14]. Unfortunately many data privacy laws have resulted in requiring users to agree to accept more responsibility of the security of the data being collected. In the case of IoT devices, consumers are the ones responsible for keeping their devices updated and secure [15]. Many IoT manufacturers stop offering security updates a couple years after product release or sooner, leaving users unknowingly at risk to cyber attacks.

IoT security concerns are not unique to individual households. The United States Air Force is one of many organizations supporting the adoption of IoT devices across their systems. In 2018, the United States Air Force (USAF) launched a smart base test pilot program with AT&T to enhance Maxwell Air Force Base operations with IoT devices [16]. As the Air Force begins to include IoT devices in operational

networks, it is important to know how the devices affect the overall organizational security posture.

In 2017, attackers gained access to a casino's customer data via a vulnerable fish tank smart thermometer [17]. Through wifi connectivity, the vulnerable smart thermometer provided attackers a path into the casino's private network, exposing private customer information.

Mobile devices share many characteristics of IoT devices, but the way users interact with their smartphones sets them apart. Additionally, smartphones typically have less variety when it comes to operating systems and configurations. In the field of mobile security, examiners are mostly concerned with data leakage. The amount of valuable information that is found on a smartphone is astounding and many users are unknowingly giving away their data when installing certain mobile applications. This section discusses several mobile devices and applications that posed a serious threat to their users.

Huawei, the number one telecom supplier and second largest phone manufacturer in the world [18], produces a variety of mobile devices, equipment, and services. In May of 2019 Trump banned the use of Huawei products within the United States government, on the premise that they form a threat to national security [18].

In December of 2019, the Defense Information Systems Agency (DISA) recommended that DoD employees should not use the Chinese-owned application TikTok [19]. The United States Army, shortly thereafter, banned troops from downloading the application on any government phone. Many suspect that the application exports user data to Chinese constituents.

Facebook, the largely popular social media platform, has also been plagued with data privacy scandals. The amount of data they collect on their users and how they share that with other applications has prompted United States (US) Congress to

question their CEO's data privacy practices. Outside organizations have abused the Facebook Application Program Interface (API) to gain unauthorized access to private Facebook user data [20]. For example, Cambridge Analytica, collected over 87 million people's personally identifiable information, via a seemingly benign quiz application, Thisisyourdigitiallife [20].

User data leakage threatens the security of the United States and its citizens. From the Department of Defense (DoD)'s perspective, user data leakage poses a threat to classified programs and missions. Security experts need to know what data is being stored on government devices in order to properly classify them. Knowing what data is stored on devices requires a methodology to determine the format in which applications store user data. This research proposes the Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) methodology to determine the format of user data stored by Android applications.

1.2 Research Objectives

The goal of this research is to demonstrate that dynamic binary instrumentation tools combined with static analysis tools can effectively be used to determine the data format of popular Android applications.

1.3 Methodology

The initial steps of this research focus on understanding existing mobile data analysis techniques and tools. The second step develops a methodology that explores unique implementations of existing mobile analysis tools to allow for a dynamic approach to reverse engineer the formats of data stored by mobile applications. The effectiveness of this methodology is demonstrated through the analysis of content and format data from the following Android applications:

1. Hago Games [21]
2. August SmartLock [22]
3. Samsung SmartThings [23]
4. Garmin Connect [24]
5. Whats App [25]
6. Instagram [26]
7. Ludo King [27]
8. Viber [28]
9. Tinder [29]
10. TextNow [30]
11. WPSOffice [31]
12. Harmony [32]
13. Wink [33]
14. Tile [34]
15. Yale Connect [35]

The methodology for this research can be broken into six main objectives: device setup, initial analysis, class analysis, synthesis, and validation.

1.4 Summary

The rest of this document is broken down into four chapters: background research, methodology, results and conclusion. The background covers Android internals, the reverse engineering process, a background in forensics and related work to provide the background necessary to adequately understand this research and the value it brings to the security of mobile and IoT communities. The DBIMAFIA methodology covers device setup, the initial analysis of the Android application package (APK) and application files, native library hooking, class analysis, and synthesis and validation of results. In the results, we apply the DBIMAFIA methodology to 15 Android applications and discuss the results of the analysis. The conclusion summarizes the major conclusions of this research and discusses potential future work.

II. Background and Literature Review

This chapter provides readers with the necessary context to understand the Android operating system and how examiners can identify, extract and analyze data from its file system. This information is invaluable to understanding the Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) methodology and its application in Chapter IV.

The background of this research is broken into four subsections: Android internals, reverse engineering, forensics, and related work. The Android internals section gives a brief overview of the operating system and related components. The reverse engineering section details existing static and dynamic Android reversing tools and techniques, as well as methods for deciphering unknown file formats. The forensic section gives readers the understanding necessary to see how the results of this research contribute to the mobile forensics community. The related work section discusses research and tools foundational to the DBIMAFIA methodology.

2.1 Android

This section provides a fundamental understanding of the Android Architecture, the design of Android applications and how Android applications store and transmit examiner data with an emphasis on databases and Java enterprise beans. This foundation is necessary to understanding how examiners can reverse engineer applications. More background details can be reviewed in Google's Android developer documentation [1].

2.1.1 Android Architecture

Android is an open source Linux-based operating system, designed for numerous device types. Figure 1 breaks the platform into six major components: Linux kernel, Hardware Abstraction Layer (HAL), Android Run-Time (ART), Native C/C++ Libraries, Java API framework, and System applications [1].

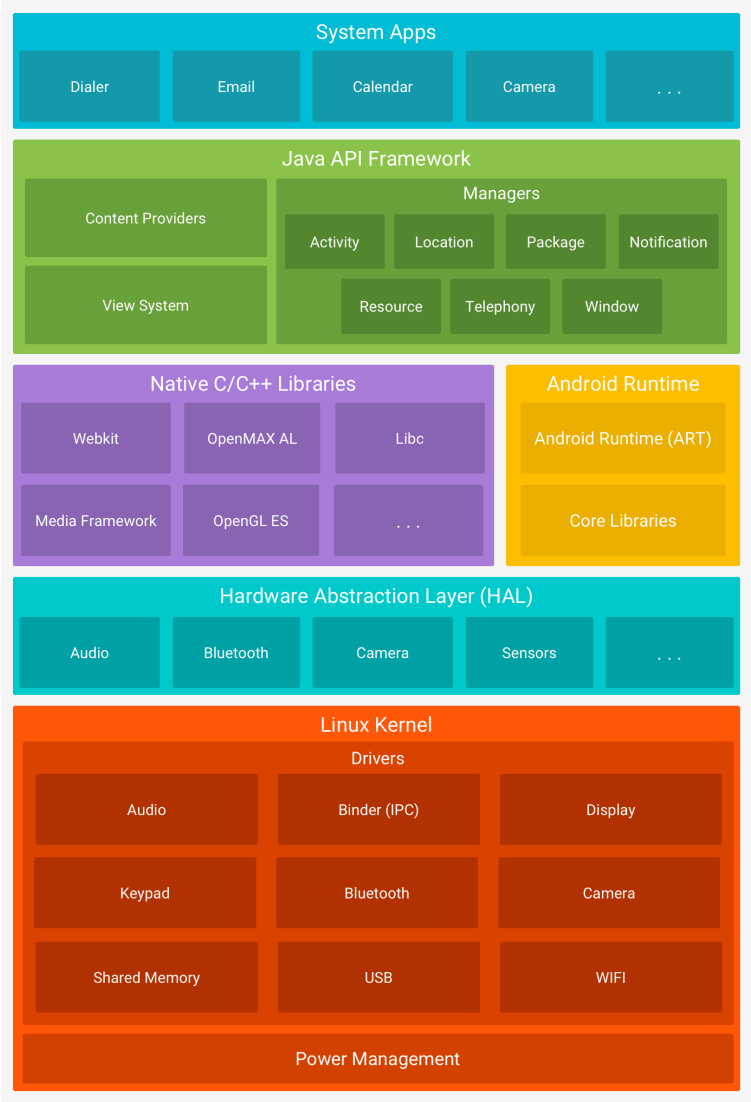

Figure 1: The Android software stack [1].

The Linux Kernel provides the foundation of the Android platform. The HAL

provides standard interfaces for the higher level Java Application Program Interface (API) framework to communicate with hardware. Android run-time implements the API that runs each application in its own process with its own instance of Android Run-Time (ART) [1]. ART runs at least one virtual machine for each application by executing Dalvik Executable (DEX) files, a byte code format meant to optimize memory usage [36]. Native C/C++ libraries use numerous Android system components. These libraries implement many of the essential functionalities required by the operating system and third party applications. The Java API framework exposes the native libraries to higher-level applications. Lastly, the system applications provide examiners interfaces for examiners to browse the web or message others, while also allowing these basic functionalities to be built into third-party applications.

2.1.2 Android Application Fundamentals

Before a examiner can begin to reverse engineer an Android Application, he or she must first understand the Android application components and how they work together. This section explains the Android Application architecture.

All Android applications are written in Java, Kotlin, or C++. The Android Software Development Kit (SDK) compiles application code with any data and resources into an Android application package (APK) [1]. The APK contains all of the contents of the application. Each application is isolated into its own security sandbox and is treated as a different Linux examiner. By default every application is run as its own process. The Android operating system implements the principle of least privilege to ensure each application has access to components needed to run the application.

Application Components are the building blocks of the application. There are four major components: activities, services, broadcast receivers, and content providers [1] [1]. As displayed in Figure 2, these components reside in their own Dalvik virtual

[1]Note some examiners include a fifth category for permissions as shown in Figure 2

machine while the application executes.

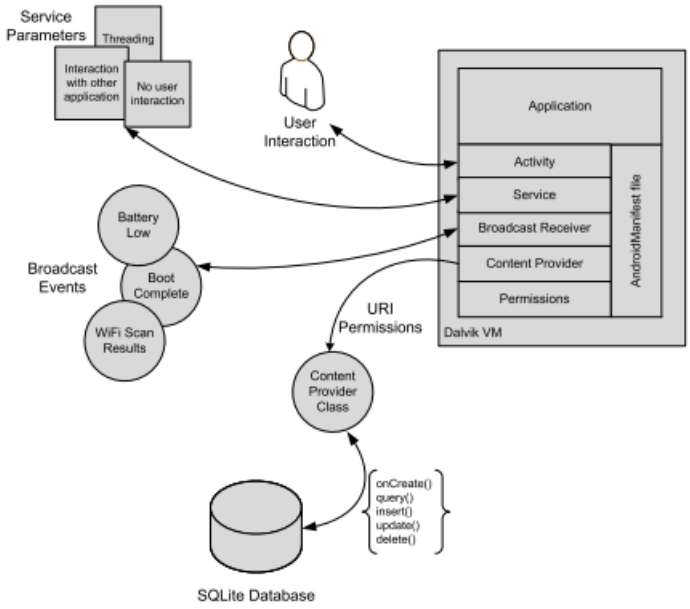

Figure 2: Android Application Components [2].

Activities serve as entry points for examiner interaction and represent a single screen or examiner interface that facilitates a number of interactions between the system and the application [1]. Activities keep track of what information is being displayed on the screen to ensure related processes continue to run. Activities also keep track of previously closed processes and their states to allow those processes to be restarted. Lastly, activities facilitate the closing of processes and the sharing of information across applications.

Services provide a method to keep an application running in the background of another application [1]. For example, a music player streams music while an examiner interacts with another application. The music service is not necessarily displaying any examiner interface, but is still required to run to provide audio to the examiner. In general there are two types of services: services the examiner is aware of and services hidden from the examiner. The system is not likely to kill services the examiner is

aware of, but there are circumstances where the system kills non-vital background services to free up Random Access Memory (RAM) for other components. The job scheduler class is used in Android 5.0 and later to schedule services and actions in an efficient manner.

Broadcast receivers enable the system to deliver events to the examiner outside the typical examiner flow [1]. This allows the application to respond to system-wide broadcast announcements, even when the application is not actively running. A good example is an application scheduling an alarm to post a notification to the examiner. When the broadcast from the system is sent, the application's broadcast receiver sees the broadcast and initiates the alarm. Android implements a broadcast receiver as a subclass of the broadcast receiver class, and delivers each broadcast as an intent object.

Content providers manage shared sets of application data that can be stored in any persistent storage location that your application can access [1]. For example any Android application with proper permissions can query the content provider of the application's contacts to read and write data. From the system's perspective a content provider is an entry point into an application for publishing named data items, identified by a Uniform Resource Identifier (URI) scheme. An application can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities which allows access to the data. Content providers allow the system to access application data without the application running and implement a fine-grained security model for application data. Content providers also can handle reads and writes to private, non-shared application data. A content provider is implemented as a subclass of Content Provider and must implement a standard set of APIs that enable other applications to perform transactions.

The Android operating system allows an application to start another application's

components. For example, a messenger application could request the native camera component to take a picture to send to another examiner within the messaging application. Android applications do not have a single entry point. To activate a component in another application, a message specifying the intent to start a component is sent to the system. Given the application has the rights to use that application's component, the system activates the requested component and the image taken is shared with the messaging application.

The manifest file informs the system what components an application contains. Your application must declare all of its components in `AndroidManifest.xml` file at the root of the application's project directory [1]. The manifest also declares to the system the permissions the application requires. In addition, it declares hardware and software features used and sets the minimum API levels and API libraries required to run the application.

2.1.3 Android Databases

Understanding how developers store application data is vital to deciphering decompiled code and determining the format of unknown file types. Databases are the obvious choice when deciding to store data that needs be accessed by multiple examiners. By definition, a database is an electronic system that allows data to be accessed, manipulated, and updated [37]. This section focuses discussion on relational databases, and object-oriented databases within the context of Android applications.

Java's object-oriented nature was not designed with relational databases in mind. Data stored in objects must be simplified in order to conform the scalar only format of typical relational database requirements. Developers must query the relational database, manipulate the data and then store that data into an object. Figure 3 is a simple `C#` code example of querying data from an SQL database, reformatting and

saving the data to the variable, name. These types of queries can usually be identified by searching the code for mysql related commands in the decompiled code.

```
var person = "SELECT key, first_name, last_name, sex, age FROM
   persons WHERE id = 1";
var result = context.Persons.FromSqlRaw(person).ToList();
var name = result[0]["last_name"];
```

Figure 3: Without object relational mapper [3].

Object relational mappers attempt to reduce this burden on developers. Object relational mapper APIs allow the developer to call for the object, and the mapper makes all the necessary queries to get the object from the relational database. This comes at a cost of efficiency, as the data still needs to be manipulated every time data is transferred to and from the database. Figure 4 is an example of the use of an object relational mapper API, and how the query and data manipulation is abstracted from the developer.

```
var person = repository.Get_person(1);
var first_name = person.Get_first_name();
```

Figure 4: With object relational mapper [3].

Object oriented databases seek to reduce the processing burden of the typical database formats by storing the data in object form. This eliminates the need to manipulate and reformat data as it is being read from and written to the database. XML and NoSQL databases both support saving data in this form [4].

NoSql databases are unique in the sense that they support unstructured storage. As illustrated in Figure 5, this means fixed table structures are not required in NoSql. These flexible key-value pair based structures allow databases to be schema-free or blue print free.

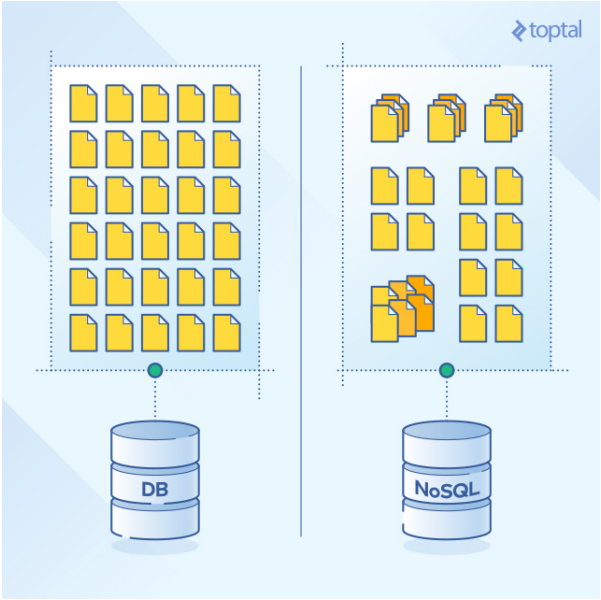

Figure 5: Relational vs NoSQL [4].

In addition to storing data in object format, NoSql databases can also use column, document, key value and XML store formats [4]. Despite their schema free nature, NoSql databases still require application or type specific database parsers to properly view the data within.

Ultimately developers have plenty of options. Databases are popular, but data can be stored in other formats, potentially using other open source or proprietary file formats.

2.1.4 Enterprise Beans

This section provides an overview of enterprise beans and its sub-types. Understanding enterprise beans is important to this research because they are commonly implemented in conjunction with various data serialization libraries. For more detailed documentation on enterprise beans, refer to Java's Enterprise Edition 6 documentation [38].

An enterprise bean is a server-side component that encapsulates the business logic of an application, which is the code that fulfills the purpose of the application. Enterprise beans simplify the development of large applications by putting beans into Enterprise Java Bean containers (EJBs), keeping logic within the bean, and delivering portable components [38]. EJB containers provide system-level services to the bean. This allows the bean developer to focus on business problems, while the containers handle system-level services. The client developer no longer has to code the routines that implement rules or database access, because the bean contains this logic. The application assembler can now build applications with these portable components.

Enterprise beans support interoperability; they can run across multiple devices, while keeping their location transparent to clients. Additionally enterprise beans support transactions, allowing concurrent access to objects, but maintaining data integrity. Enterprise beans come in two types: session beans and message-driven beans [38].

2.1.4.1 Session Beans

A session bean encapsulates business logic that a client can invoke locally or remotely. To access an application on a server, the client invokes the session bean's methods. The session performs the work for its client, shielding it from the complexity by executing tasks inside the server [38]. Note that session beans do not save data to

a database and are therefore not persistent.

Session beans come in three types: stateful, stateless, and singleton. Table 1 explains when each type of session bean is necessary. A stateful session bean contains the state of an object using instance variables that represent the state of a unique client-bean session. This state is often referred to as the conversational states as the client talks with its bean. When the client removes the bean, the session ends and the state disappears. Stateless beans do not maintain a conversational state with the client. When a client invokes methods of a stateless bean, the bean contains a client-specific state for the duration of that invocation [38]. Once the method finishes running, the state disappears. Pooling the stateless beans guarantees thread safety. Additionally stateless beans only have to be stateless with regards to the client. The private state of each bean can be held over to the next invocation. Unlike stateful beans, stateless beans can support multiple clients. Singleton session beans are instantiated once per application and exist for the entire life-cycle of the application. They are designed for situations where a single bean is shared across clients, who are concurrently accessing it [38].

Table 1: Types of Session Beans.

Session Beans	
Stateful	The bean's state represents the interaction between bean and specific client The bean needs to hold information about client across method invocations. The bean mediates between the client and other application components. The bean manages the work flow of several beans.
Stateless	The bean's state has no data for a specific client. A method invocation requires the bean to perform a task for all clients. The bean implements a web service.
Singleton	State needs to be shared across the application. An enterprise bean needs to be accessed by multiple threads concurrently. The app needs a bean to perform tasks upon app startup and shutdown. State needs to be shared across the application.

2.1.4.2 Message-Driven Beans

A message-driven bean allows asynchronous message processing between end points, which session beans cannot support. Message-driven beans act as a Java Message Service (JMS) listener. Messages can be sent by any Java EE component or JMS application or system. Message-driven beans can process JMS and a variety of other message types. Message-Driven beans have the following characteristics [38]:

1. They execute upon receipt of a single client message.
2. They are invoked asynchronously.
3. They are relatively short-lived.
4. They do not represent directly shared data in the database, but they can access and update this data.

5. They can be transaction-aware.

6. They are stateless.

Message-driven beans share a number of features with stateless session beans. Neither of the bean instances retain data for specific clients. Both beans ensure that all instances are equivalent, allowing proper pooling of beans. Lastly, both beans can process data from multiple clients.

Message-driven beans handle state differently from instance to instance. The instance variables of the message driven beans contain some state across client message handling. They do this through the use of the Java Metadata Interface (JMI) API, open database connections, or object references to enterprise beans [38]. Clients do not invoke methods directly on the beans, but rather clients access the beans through JMS or other similar protocols. They send messages to the destination and the message-driven bean class acts as the message listener. When the message arrives, the container calls the `onMessage` method to process the message. This method typically casts the message out to one of the five JMS message types and handles it according to the logic of the application. `onMessage` often invokes helper methods or other session beans to process the message and store it in a database. In short, Message-Driven beans offer asynchronous message processing that avoids tying up server resources.

2.2 Reverse Engineering Android Applications

Reverse engineering is a process of determining how a system works without access to the source code or original specifications [39]. This process supports legitimate interoperability to closed systems and illegitimate adversaries intent on gaining access to unauthorized data.

In this research, we divide reverse engineering into two categories: static and dynamic software analysis. During static software analysis, the examiner takes the executable code from the device, including the stored memory, and recreates the software structure. Static analysis helps to create a flow diagram of software, to assist examiners, and to derive the behavior and function of the code. Dynamic analysis provides the software's behavior, giving examiners insight into the volatile memory being used during execution that may not be visible during static analysis. From static and dynamic analysis, the examiner can determine the device's behavior, functions, protocols, and communication sensors of the device.

2.2.1 Static Reverse Engineering Android Applications

Application developers seldom release the source code of their applications [40]. Android application reverse engineering is necessary to understand how the application communicates and stores information. While reversing Android applications follows the traditional reverse engineering process, specific tools and techniques are necessary to handle the intricacies of Android applications.

Static analysis of an application gives the examiner a better understanding of the layout of the file system, without having to run the file [41]. Dr. Richard Dill breaks down Android application reverse engineering into five distinct steps: Access, Unpacking, Dissimilation, Building, and Signing [40]. These steps recognize the common process used to reverse engineer Android applications, however the building and signing steps are only necessary for cases when the application is modified to support other analysis techniques.

Access

In order to retrieve an APK for an Android application, the examiners must either download the APK file using websites like Apkpure.com [42] or Apkmirror.com [43] or

retrieve them directly from the phone. Examiners statically analyze the downloaded APK to determine the application's layout. However, used applications retrieved from a device provide the examiner with more data to analyze.

The Android operating system does not initially allow examiners to access their application's internal files. Linux partitions files and limits access to the internal file system. In order to retrieve these files, the examiner must first elevate privileges to root access; this can be gained by unlocking your bootloader and running an `SU` binary in the system partition [44]. When an application tries to run, the operating system then checks this `SU` binary to ensure that the application attempting to run as root has been verified to have root privileges [44]. A SuperUser management application is then used to grant privileges to applications on your system. Many rooting methods exist; some to specific hardware manufacturers, while others generically run on any device running the Android operating system. After gaining root privilieges, the examiner can use Android Debug Bridge (ADB) to access the entire file system.

Unpacking

Once the APK is acquired, the APK is unpacked to expose the application's files and DEX code. The Android operating system uses the ZLIB format to compress its applications before distributing applications via the Google Play Store [45]. Files compressed with the ZLIB compression package can be unpacked using popular tools such as 7Zip [46] or JADX [47]. Unpacking reveals the application's file system. From this, examiners have access to readable files with known file formats, however `.DEX` file(s) requires dissimilation to be further examined.

Dissimilation

Dissimilation requires the examiner to either disassemble or decompile the unpacked `.DEX` files to properly convert the `.DEX` files into meaningful information [40]. Disassembly of the unpacked `.DEX` files results in Smali code [48], an intermediary

language between the source and the byte code. Smali and Baksmali are respectively assemblers and disassemblers for the `.DEX` file format [49]. Smali code describes at a low level how and where the Android application stores information in registers, variables, methods, and various memory locations. Decompiling the byte code of the `.DEX` file results in a Java code representation, which is not the original source code, but logically equivalent. The logically equivalent Java code is another way for examiners to understand the application's code. JADX, JEB [50], and IDA Pro [51] are all popular decompilers for the dex file format.

Building

The building phase consists of reassembling the files of the application back into an APK. Android Studio and ApkTool are both free tools that can be used to build APKs.

Signing

The last step is to sign the APK before dissemination. This is done using a public key certificate and allows Google to ensure that all future updates come from the same developer. This can be accomplished using the build feature in Android Studio [52].

2.2.2 Dynamic Reverse Engineering Android Applications

This section demonstrates three effective dynamic reverse engineering techniques for the Android operating system: sandboxing, debugging, and dynamic binary instrumentation. Unlike static reverse engineering that provides a picture of the application without execution, dynamic reverse engineering allows the examiner to observe the application's behavior as it executes [41].

Sandboxing

Sandboxing allows examiners to understand how applications work without hav-

ing access to the original application's source code. Sandboxes execute applications within various types of containers that log relevant actions and changes to memory. Sandboxes are useful to detect malware and log internet requests. Cuckoo-droid [53] and Joe Sandbox [54] are both Android sandboxes that offer detailed information on applications.

Debugging

Debugging allows an examiner to set breakpoints in the code of an application to view variable types and data as they are stored on the memory heap and stack. Debugging allows examiners to dynamically watch data while it is being transformed.

Developers release Android applications without debugging enabled, requiring examiners to decompile, enable debug mode, recompile, resign, and reinstall the application before Java code debugging can occur with the native Android Studio application. This method allows examiners the ability to step through the decompiled Java code of the application. Android studio provides Android debugging functionality for free [52], but requires some version of Java source code. The Smalidea plugin [48] for Android Studio allows the examiner to step through the unpacked `.DEX` files of an application, without ever having to decompile or repackage an application [48]. Android debuggers like IDA Pro [51] and JEB [50] offer paid debugging options.

Dynamic Binary Instrumentation

Dynamic binary instrumentation provides the ability to modify application behavior at run-time. These methods inject an `agent` into the application, that allows methods and data to be changed during application execution. This can be used to display or modify variables upon entering or exiting a function as pushed on the stack.

One such tool, Frida [5], injects a Javascript engine into the application, allowing examiners to write Javascript code to interact and change code of the application

as it executes. This allows the examiner to manipulate and analyze applications as needed.

Figure 6 illustrates how Frida interacts with the target application to run custom Javascript code to modify functionality at runtime. Frida begins by saving the frida-agent shared library to the application. The Linux `Ptrace` command is used to hijack thread2. The hijacked thread writes the bootstrapper to memory, a program that creates a thread without user interaction. The bootstrapper creates the Frida thread within the program. The Frida thread loads the frida-agent into memory, and a connection is established with the debugger process to inject any code from the examiner. The hijacked thread resumes, and the debugged process runs as normal.

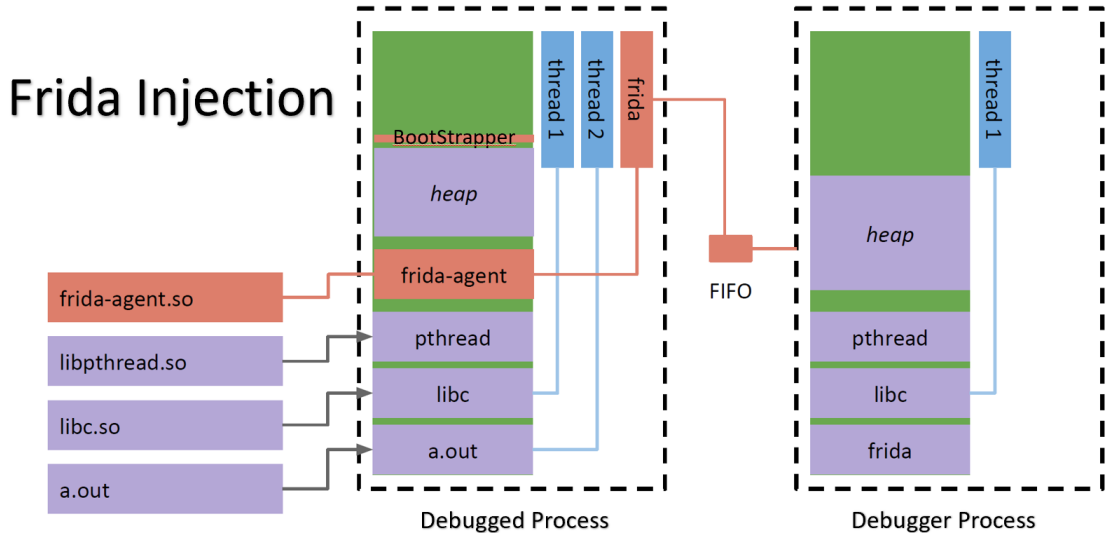

Figure 6: How Frida Interacts with Application [5].

Rather than modifying the code at rest or while debugging the code, Frida's technique of code injection modifies the code at run-time. This subverts common anti-reverse engineering techniques that look for signs of code modification or changes common to debuggers' breakpoints.

2.2.3 Anti-Reverse Engineering Techniques

Developers work to protect the application examiner data, employing a variety of techniques to thwart reverse engineering. This section focuses on methods to protect data at rest and methods to protect data during execution.

Proguard, an application obfuscator, is a free program that shrinks Android APKs, optimizes code, and obfuscates classes, methods, and variables. Proguard's obfuscation techniques rename and remove unused variables, classes and methods. This makes code analysis challenging, because variable, method and class names help determine the application's functionality. Additionally, Proguard's effect on proper decompilation can prevent functions from being properly decompiled to Java code. If the improperly decompiled code should ever be modified and re-compiled, the application is likely to fail at compilation [55].

Java-level debugging is a reverse engineering technique to help understand code flow, however all applications are by default not debuggable. An examiner can alter the flag in the application's `manifest.xml` file, but this requires the application to be decompiled, modified, resigned, and redeployed. Even with this illicit modification, application developers employ methods to detect and prevent their applications from being debugged: code hashing, time checks, Tracer Process Identity (TRACERPID) comparisons, and certification validation.

Any time a break point is set, the code is modified [56]. This change results in a modified hash of that given block of code. Concerned developers check hashes of portions of code during execution to detect and crash a debugged version of the application.

Timing checks identify delays in code execution to determine whether an application is being debugged [56]. This method is only as secure as the timing mechanism used to detect delays. A reverse engineer could spoof the timer to convince the

application that it is in fact running as normal.

The TRACERPID represents the process id (PID) of the application that is tracing the existing process. This value is normally zero, however when debugging, this value is set to the PID value of the debugger. Developers can check this value at random times during execution to detect and crash the application when it is being debugged [56].

Developers can view the certificate being used to run the application to determine if it is equivalent to the original [56]. Repackaged applications fail this check, as they are not signed with the original certificate.

Dexguard [57] and Dexprotector [58] are both commercialized programs that implement several anti-debugging techniques to keep applications safe from a variety of debugging techniques. A reverse engineer could search the decompiled code for comparison operators and remove them from the application, but is time consuming and tedious.

2.2.4 Deciphering file formats

This section details traditional static and dynamic reversing techniques to decipher file formats, as first described in Secrets of Reverse Engineering [59], Eilam.

First, the examiner should determine the purpose and intent of the application. Second, the examiner identifies known, easy to spot values in the analyzed files. This may or may not be possible depending on encryption, compression, or obfuscation levels. Third, Eliam recommends providing different inputs to monitor how the targeted file changes. The fourth step is to verify that features such as passwords are indeed checked when interacting with the application. The fifth step involves opening the file in a hex editor and checking the first couple bytes for a file signature and making observations on the rest of the file. Eliam mentions character distribution analysis as

a method of determining whether or not the file is encrypted. The sixth step focuses on creating a list of all imported functions using programs such as Windows Dumpbin [60]. This list provides an overview of how the program works and reads and writes to files. He recommends placing break points on function calls identified to the imported functions to help dynamically step through the program in a debugger. He provides a specific Windows example, examining entries in the `Kernel32.dll` and uses that to trace calls to the file input and output APIs.

2.3 Forensics

Although this research is founded on reverse engineering principles and processes, it is also necessary to understand the forensic process in order to see how the results of this research can be applied to extracting forensically relevant data. The major distinction between the two fields is that forensic results typically support the enforcement of the law and attempt to minimize changes made to a system, whereas reverse engineering attempts to understand how a system works. This section is broken into four forensic specialties: traditional, digital, mobile, and Internet of Things (IoT).

2.3.1 Traditional Forensics

Forensics is the discipline of applying scientific knowledge to legal problems [61]. Forensic science is therefore any science used for the purpose of law [61]. All forensic sciences are founded in Locard's principle, which states that "any two objects that come in contact with each other will exchange material." This leads to the conclusion that a person or object always leaves a trace when visiting a place or crime scene.

It is upon this theory that the Organization of Scientific Area Committees (OSAC) developed a framework for harmonizing forensic science practices. OSAC's framework divides the forensics process into four distinct parts [62]:

1. Authentication: Verify the claim.

2. Identification: Associate the entity with the action taken.

3. Classification: Compare different pieces of evidence to identify specific origin(s).

4. Reconstruction: Put the evidence together to answer the who, what and where.

Each step should be a repeatable and consistent process that makes up the greater forensics framework. All four parts act as individual forms of evaluation that help inform and refine the other parts of the forensics process. OSAC determined that the value of forensic science was its use of scientific reasoning to provide decision-makers with an understanding of evidence presented to help them make educated decisions.

2.3.2 Digital Forensics

Digital forensics includes everything from physical electronic devices to network traffic and various digital media forms. Locard's principle states that any interaction with an electronic device leaves a trace. On a relatively small system, examiner interaction, programs running in the background and logging of file system changes add up to hundreds of gigabytes of digital forensics data. Locating evidence that is relevant to a case among the billions of other bytes of information can be challenging.

OSAC's framework applies to all forensic processes, including those outside the realm of electronics. It is upon this framework that many digital forensic processes are developed. While digital forensic processes vary greatly by situation, the four principles that ring true across most processes are [63]:

- No action should be taken that changes data that may later be used as evidence.

- Any examiner accessing original data should be competent and able to explain necessity and implications of actions taken.

- Steps taken should be repeatable and properly documented such that another examiner could reach the same results.

- Reconstructions: Put the evidence together to answer the who, what and where.

With these principles in mind many digital forensic investigations begin by making a bitwise copy of the contents of a computer's hard drive. Following this a capture of the system's memory is made. Most examiners then perform analysis on the copied drives, and memory capture, ensuring analysis tools do not affect the original drive or system. Please note that most memory capture tools and some hard drive copying methods leave at least some trace on the machine; it is therefore imperative that examiners are knowledgeable of exactly what traces are left from their activity on the original drive.

2.3.3 Mobile Forensics

Mobile forensics is a subset of digital forensics that frequently refers to the examination of smart phones [64].

Mobile forensics would ideally follow the same process of imaging the contents of a smart phone's hard drive and performing a capture of the phone's memory. Unfortunately, forensics on mobile phones is difficult because of safeguards phone manufacturers, carriers and developers have put in place. Open source and commercial mobile forensic programs rely heavily on kernel or root access to a device to function properly. The ideal of not changing a single bit on a mobile device during the forensics process is therefore not practical [65]. This has lead many to be skeptical of the forensic soundness of mobile forensic techniques. However, with proper technical expertise, it is possible to extract valuable evidence from mobile devices [66]. To do so, the examiner must ensure that they can properly explain the necessity and implications of any actions taken. Mobile security risks can be broken into three

risk categories: physical, service, and application [67]. These risks are assessed to ensure that evidence on mobile devices has not been altered before, during, or after the collection of devices.

2.3.4 IoT Forensics

An IoT device is defined as any embedded device with the capability of remote connectivity. The examination of IoT devices crosses several forensic disciplines. There is onboard memory on most IoT devices that typically stores firmware and small amounts of log data. There are also numerous mobile companion applications that help set up, control and store varying amounts of IoT data.

Accessing IoT data from these locations requires the examiner to interact with the device's onboard memory via physical acquisition of a JTAG port. Other methods require examiners to desolder the memory chip and to solder it onto an identical device. These methods alter the state of hardware and carry severe risk of destroying potential evidence.

IoT mobile applications have the potential to store valuable IoT data but require examiners to gain kernel or root privileges to access most data. Rooting common mobile devices requires a device password, and an available exploit to utilize a vulnerability in a specific version of hardware and software. Despite these difficulties experts focus many investigations on mobile devices, because they frequently contain large amounts of pertinent examiner data from a variety of sources.

2.4 Related Work

Related work is broken into six categories of tools and research: Disassemblers and Decompilers, Static Instrumentation Tools, Dynamic binary instrumentation Tools, Application Analysis, Data Collection Tools, and Android File Format Analysis. All

six categories of tools and research are closely related to developing a methodology to determine the format of Android application examiner data.

In support of the United States Air Force Research Labratory (AFRL), TwoSix Labs released a research paper in December 2019, documenting and comparing popular disassembly and decompilation, static instrumentation, and dynamic analysis tools [6].

2.4.1 Disassemblers and Decompilers

TwoSix Labs analyzed and compared IDA Pro [51], Ghidra [68], and Binary Ninja [69]. These disassemblers and decompilers were presumably chosen for their popularity amongst the reverse engineering community. All three programs support disassembly of various architectures, but Binary Ninja currently lacks decompilation capabilities. Both Ghidra and IDA Pro decompile most binaries to a C like pseudocode that can aid examiners in reverse engineering binaries from a variety of architectures. Binary Ninja's lack of decompilers is supplemented with a variety of intermediate languages that focus more on readability of code, rather than reproducing source code. TwoSix labs found all three tools to offer advantages in particular use cases. Figure 7 summarizes the features of all three tools.

2.4.2 Static Instrumentation Tools

TwoSix Labs analyzed three static instrumentation tools that facilitate debugging, tracing and profiling [6]. Multiverse [70] is a static rewriter that can add or remove security features of a binary without debug symbols or relocation entries. Multiverse does so by disassembling the binary into a superset disassembly that contains all legal instructions and then uses an instruction rewriter to relocate instructions and modify control flow. Note that while Multiverse's core functionality does not require

	IDA Pro	Ghidra	Binary Ninja
Reference Link	https://www.Hex-Rays.com/	https://ghidra-sre.org	https://binary.ninja
Target Type	Binary	Binary	Binary
Host Operating System	Linux; macOS; Windows	Linux; macOS; Windows	Linux; macOS; Windows
Target Operating System	**Windows:** PE (Portable Executable); MS DOS/MS DOS Driver/MS DOS Com; Windows Crash Dump; etc. **macOS/iOS**: Mach-O; etc. **Linux:** ELF; etc. **Android:** DEX Format; etc. **Other**: JAR Format; COFF (Common Object File Format); Raw Binary; etc.	**Windows:** PE (Portable Executable); etc. macOS/iOS: Mach-O; etc. **Linux:** ELF; etc. **Android:** DEX Format; etc. **Other:** COFF (Common Object File Format); Raw Binaries; etc.	**Windows:** PE (Portable Executable); etc. macOS/iOS: Mach-O; etc. **Linux:** ELF; etc. **Other:** COFF (Common Object File Format); Raw Binary; etc.
Host Architecture	x86 (32, 64)	x86 (32, 64)	x86 (32, 64)
Target Architecture	**IDA Pro Disassembler:** x86 (16, 32, 64); ARM (32, 64); PPC (32; 64); MIPS (32, 64); SPARC (32, 64); PIC (12, 16, 17, 18, 24); Java bytecode; DEX bytecode; etc. **Hex-Rays Decompiler:** x86 (32, 64); ARM (32, 64); PPC (32; 64); MIPS (32, 64); etc.	**Disassembler and Decompiler:** x86 (16, 32, 64); ARM (32, 64); PPC (32; 64); MIPS (32, 64); SPARC (32, 64); PIC (12, 16, 17, 18, 24); Java bytecode; DEX bytecode; etc.	**Dissassembler:** x86 (32, 64); ARM (32, 64); MIPS (32, 64); Thumb3; PowerPC; 6502; etc...
Initial Release	Disassembler Release (Commercial): 1996 Decompiler Release: 2007	March 2019	2016
License Type	Proprietary	Open-Source	Proprietary
Maintenance	Maintained by Hex-Rays SA	Maintained by the National Security Agency (NSA)	Maintained by Vector 35

Figure 7: Dissassemblers/Decompilers Comparison [6].

instrumentation, some of its more advanced capabilities require instrumentation to be present. DDisasm [71] aims to provide examiners with a tool that can disassemble executables into a form that can be correctly assembled back into an execuatble in a fully automated manner. In tests DDisam correctly reassembled 98.44 percent of binaries. Note that both Multiverse and DDISAM were designed specifically for the x86 architecture, and not Android architectures. Library to Instrument Executable Formats (LIEF) was the last static instrumentation tool mentioned and it was mentioned because of its amount of documentation and ease of use. It offers Python, C++, and C APIs to parse, edit and analyze executables. LIEF works with Linux, macOS, Windows, and Android x86 binaries. Figure 8 summarizes the features of all three tools.

	Multiverse	**Ddisasm**	**LIEF**
Reference Link	github.com/utds3lab/multiverse	github.com/GrammaTech/ddisasm	lief.quarkslab.com
Target Type	Binary	Binary	Binary
Host/Target Operating System	Linux	Linux	Linux; macOS; Windows; Android (target only)
Host/Target Architecture	x86 (32, 64)	x86 (32, 64)	x86 (32, 64)
Initial Release	February 2018	April 2018	April 2019
License Type	Open-Source	Open-Source	Open-Source

Figure 8: Static Instrumentation Tools Comparison [6].

2.4.3 Dynamic Binary Instrumentation Tools

TwoSix Labs compared performance and use cases between two dynamic binary instrumentation tools: rr [72] and Frida [51]. The rr open source x86 debugger enables examiners to deterministically replay execution and reverse execute from a breakpoint [73]. The goal is to provide examiners with the ability to retrigger crashes. Frida,

as discussed in the background of this research, is a dynamic binary instrumentation toolkit designed for x86 and ARM architectures. TwoSix Labs found Frida's ability to inject code and trace execution to enable new methods of reverse engineering. Figure 9 summarizes the features of both dynamic binary instrumentation tools.

	rr	Frida
Reference Link	https://rr-project.org/	http://www.frida.re/
Target Type	Binary	Binary
Host/Target Operating System	Linux; macOS; Windows;	Linux; macOS; iOS; Windows; Android; QNX
Host/Target Architecture	x86 (32, 64)	x86 (32, 64); ARM (32, 64)
Initial Release	March 2014	Dec 2013
License Type	Open-Source	Open-Source

Figure 9: Dynamic Binary Instrumentation Tools Comparison [6].

2.4.4 Application Analysis

Sicurezza Reti explored how Java method hooking could dynamically check applications for malicious code or vulnerabilities [36]. His research successfully identified private examiner information accesses, internet connections, and deprecated or insecure protocol usage. His research utilized Android Dynamic Binary Instrumentation (ADBI) and Legend [74] to hook methods and analyze 32-bit applications running on Android versions 4.2-6.01 [75].

2.4.5 Data Collection Tools

In June of 2018, Ricordo Spolaor published research on DELTA, a data extraction and logging tool for Android [76]. He categorized smartphone data into three

categories: sensor data, device context data, and examiner interaction. Spolaor mentioned Android data collection tools: Systemsens [77], DroidWatch [78], Mobilsens [79], PhoneLab [80], LiveLab [81], and DeviceAnalyzer [82]. He stated that these tools fail to provide a modular design, consistency in data collection, and consistency in sampling rates. Spolaor argued that DELTA meets these requirements and claimed that DELTA collected data from more sources than any of the aforementioned tools.

DELTA and the five other tools mentioned in Spolaor's report lack any support for extraction of data from application specific databases. Andriller [7], an open source forensic decoder, offers its examiners decoders for Android applications. Figure 10 shows the decoders available through Andriller. The tool focuses on the extraction of text and voice messages from the Android operating system and popular Android messaging applications. This research found Andriller to be the best open source Android forensic tool that extracted examiner data from applications. Although it focused specifically on voice and messaging applications, its interface, feature set, and modularity set it apart from other Android forensic tools.

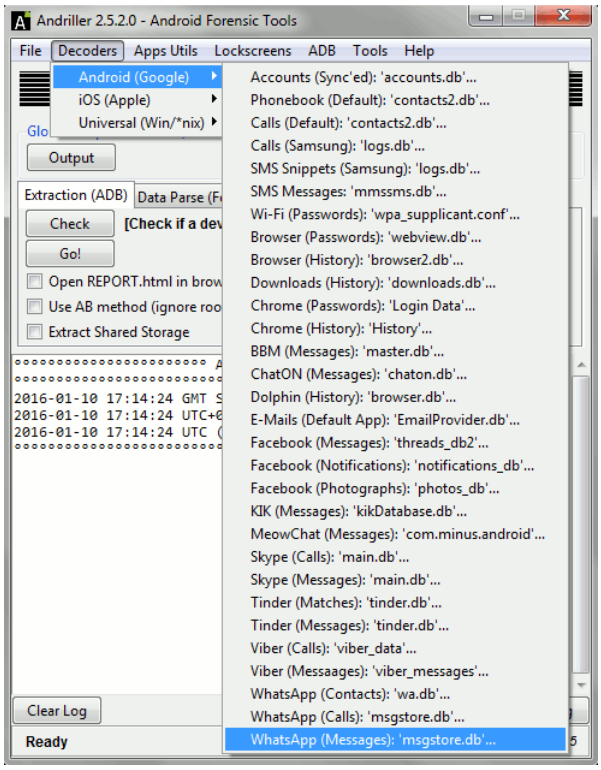

Figure 10: Andriller's Decoders [7].

2.4.6 Android File Format Analysis

Dr. Richard Dill's research explored the automation of mobile device file format analysis. Dr. Dill parsed and prepped applications to display proper method offsets. He created and ran the Automated Data Structure Slayer (ADSS) to automate the injection of hooks to uncover structures used to store and process application data.

2.4.7 Related Work Summary

This section discussed six categories of tools and research that are foundational to this research. The DBIMAFIA methodology builds off of Dr. Dill's and the aforementioned tools and research to provide examiners with a way to approach mobile data format analysis that does not require the parsing and prepping of an application.

2.5 Summary

This chapter covered Android internals and the reverse engineering process, provided a background in forensics and related work necessary to understand this research and the value it brings to the security of mobile and IoT communities.

III. Methodology

In this research we debut the Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) methodology for analyzing formats of mobile application data. The DBIMAFIA methodology covers device setup, initial analysis of the application and its files, native method hooking, class analysis, and the synthesis and validation of results. Step one of DBIMAFIA focuses on gaining root access. Step two primarily focuses on application file analysis and unpacking and decompilation of the app's Android application package (APK). Step three of DBIMAFIA explores how dynamic binary instrumentation tools can be used to hook native method calls. Step four uses static and analysis and dynamic binary instrumentation to analyze Java classes. Step five synthesizes the findings from the class analysis phase to come to conculsions about how the application is storing and manipulating user data. The sixth and final step uses a byte debugger to determine the validity of conclusions made in step five.

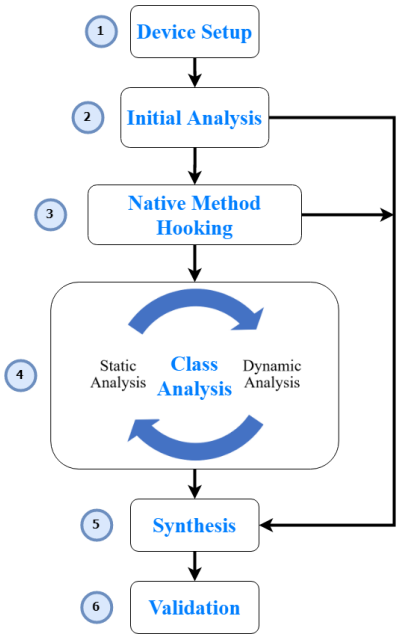

Figure 11: DBIMAFIA Methodology Overview.

3.1 Device Setup

The DBIMAFIA methodology requires elevated access to the mobile phone's internal file system to determine how data is processed by the application. In step one of the DBIMAFIA methodology, the examiner gains root access to the mobile device to ensure he/she has access to necessary files.

The examiner downloads and installs the Android SDK platform-Tools package to a computer [1]. This package contains Android Debug Bridge (ADB), a command line tool that interacts with the Android device. The examiner executes the following steps to achieve root access [83]:

1. Enable Developer Options and Unlock Bootloader.
2. Install Magisk Manager.
3. Push factory boot image to device.
4. Patch the boot image
5. Restart and flash phone with modified boot image.

Once the phone is rooted, the examiner installs Frida server to support dynamic analysis [5].

3.2 Initial Analysis

The second step in the DBIMAFIA process is responsible for the analysis of files changed due to examiner interaction and the unpacking and decompilation of the application's APK. Initial Analysis is broken into six subsetps:

1. Logical copy of application and APK

2. Application interaction

3. File analysis

4. Unpack and decompile/disassemble APK

5. ID libraries and application architecture

6. Modify Frida for 32-bit support

3.2.1 Initial Analysis Step One: Logical Copy of Application files and APK

The examiner pulls a copy of the APK and stored application data in the `\data\data\com.appname.example\` directory, before dynamically interacting with the application. This gives the examiner a way to come back to a fresh state between each test and supports the need for file comparison in the initial analysis phase.

The next step is recording the time that interaction with the application began. This research found that creating an empty file immediately before application interaction creates a reference point to identify future file changes.

3.2.2 Initial Analysis Step Two: Application Interaction

At this point the examiner opens and interacts with the app, testing each feature to identify how data is stored. For applications requiring external hardware, this step requires the examiner to interact with the external hardware. Next, the examiner notes the file changes since the creation of the blank file. The "-newer" option of the Linux `find` command provides examiners with a way of recording recently modified files.

Examiners run Modification Detective, a script found in Section 1.1 of the Appendix, to automate the majority of this process. User-input for both the application

and external devices is still required. Modification Detective creates a blank file in the \sdcard folder and waits a specified number of seconds for user interaction. Files that change while the script is running are pulled back to the examiner's computer.

3.2.3 Initial Analysis Step Three: File Analysis

In this step, the examiner analyzes the structure of the APK via static analysis. The examiner begins with using the Linux `diff` command to compare the modified files with the files saved prior to interaction. The Linux `file` command is then used to determine the file type of the modified file(s). Linux's `binwalk -E` is used to output an entropy graph that examiners can use to determine whether the file is obfuscated or encrypted. Examiners look for linear entropy graphs to identify encrypted files. The Linux `strings` command is run on the file of interest to identify American Standard Code for Information Interchange (ASCII) strings. If examiners come across plaintext or known file types, they should skip to step 5 of the DBIMAFIA methodology and summarize their findings.

3.2.4 Initial Analysis Step Four: Unpack and Decompile APK

As discussed in Section 2.1.1, applications are distributed in a compressed package known as an APK. To analyze an application, the examiner must unpack the APK into `.DEX` and application files. The `.DEX` files are then disassembled and decompiled into Smali and Java pseudocode. JADX automates this process; it takes an inputted APK, unpacks, disassembles and decompiles the application for the examiner [47].

3.2.5 Initial Analysis Step Five: Identify Imported Libraries and Application Architecture

Examiners open the \lib directory inside the unpacked APK and analyze the libraries within. The folder name within the \lib directory distinguishes the application's architecture. The examiner now searches the names of the shared library files to determine the associated library names.

3.2.6 Initial Analysis Step Six: Modify Frida for 32-bit Applications

In order to get access to contents of the stack on 32-bit applications, Frida's thread and context pointers must be manually modified. These modifications ensure that the DebugStackVisitor class in the hook script properly hooks the targeted application methods. The DebugStackVisitor class extends the ArtStackVisitor class in the Android.js file of the Frida-Java environment. The ArtStackVisitor class utilizes Native Android APIs to display information on the memory stack.

The android.js file within the \lib folder of the Frida-Java-Bridge environment [5] is modified. Changes are made to the ArtStackVisitor class to adapt it for the 32-bit version of the libart.so. This modification adds offsets to the thread and context pointers to ensure they address the proper portions of memory. Examiners calculate the offsets using a modified version of Oleavr's art-internals probe.py script [84]. The original script found offsets for the virtual machine and instrumentation. Modifications to the script are made to calculate the thread and context offsets of the desired architecture and Android version. Android's open source nature allows users access to the operating system's C object files. The script accepts a C++ source code file and modifies all private methods to public. Then, the script uses C's offsetof() macro [85] to calculate the offsets of the thread and context fields.

The following changes are made to the android.js code of the Frida-Java-Bridge

environment:

As demonstrated in Figure 12, the line calculating the thread and context pointers is replaced by two lines that add offsets to the thread and context pointers.

```
//api['art::StackVisitor::StackVisitor'](visitor, thread, context,
//   WalkKind[walkKind], ptr(numFrames), checkSuspended ? 1 : 0);
visitor.add(4).writePointer(thread); //thread, of type pointer
visitor.add(40).writePointer(context); //context, of type pointer
```

Figure 12: Offsets added to thread and context pointers.

Additionally the line in Figure 13, that imports the 64-bit frame information is removed.

```
/* this._getCQFIImpl =
   api['art::StackVisitor::GetCurrentQuickFrameInfo']; */
```

Figure 13: Remove 64 bit frame info import.

These changes allow examiners to use the `ArtStackVisitor` class to view the contents of the memory stack of 32–bit Android applications.

3.3 Native Method Hooking

The third step of the DBIMAFIA methodology is responsible for the hooking of the native `libc.so open()` and `write()` methods to provide the examiner with knowledge about where to begin class analysis. This section is broken into three substeps:

1. Hook open and obtain file descriptor

2. Hook write and dump stack, and written methods

3. Analyze written data

3.3.1 Native Method Hooking Step One: Hook Open Methods and Obtain File Descriptor

The examiner must hook the `open()` and obtain the file descriptor prior to hooking the `write()`. There are multiple native libraries that read, write, and transform files, but the `libc.so` native library is the most common. The `libc.so open()` method takes in three arguments: file path, flags specifying read or write permissions, and a mode that details which user groups have what access [86]. Figure 14 illustrates an `open()` call hook that takes in the file path and compares it to the file path of interest. When they match, the file descriptor is saved for later comparison in the `libc.so write()` hook.

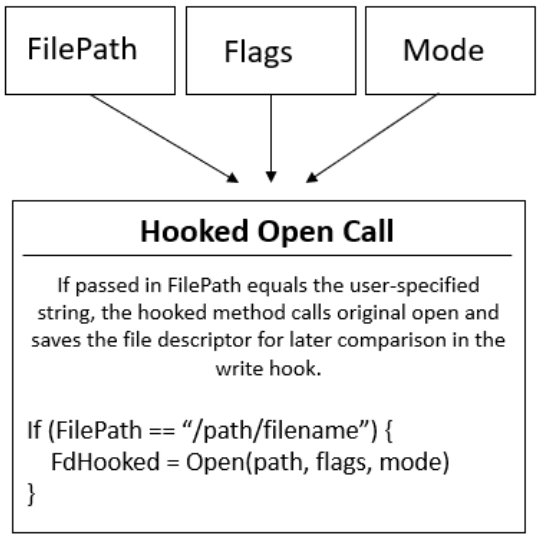

Figure 14: Obtaining File Descriptor.

The examiners use code in Figure 15 to hook the `libc.so open()` method and capture the file descriptor. As we recall from Figure 14, `libc.so open()` accepts the file path, the flags and the mode as arguments.

```
open_hook =
   Interceptor.attach(Module.findExportByName('libc.so','open'),{
onEnter: function (args) {
    var value = Memory.readUtf8String(args[0]);
    if (typeof value !== 'undefined') {
        /* If the target file name is found in the arguments, save the
           file name.*/
        if (value.indexOf(target_filename) !== -1)
                this._open_fileName = value;}
},
onLeave: function (retval) {
    // If valid file descriptor, add to file _descriptor_array
    if (retval.toInt32() > 0) {
        file_descriptor_array[retval.toInt32()] = this._open_fileName;}
}});
```

Figure 15: Javascript hook of `libc.so open()`.

3.3.2 Native Method Hooking Step Two: Hook Write, Dump Stack and Written Data

As shown in Figure 16, the examiner hooks the `libc.so write()` method to dump the memory stack prior to writing data to the target file. If the `libc.so open()` or `write()` methods are not hooking as expected, open up the Linux man page [86] and explore other methods involved with saving data. `Libc.so's iotcl()`, `fnctl()`, `write64`, `pwrite64`, `pwrite`, `writev`, and `put()` are other native candidate methods to hook.

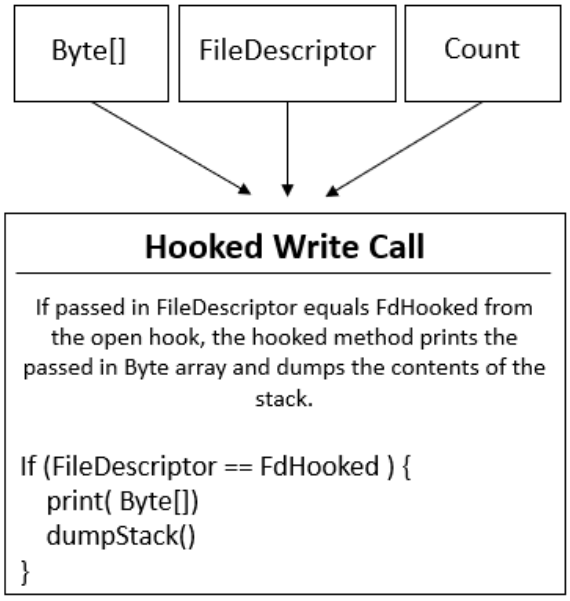

Figure 16: Dumping Stack.

As shown in Figure 16, the `libc.so write()` hook grabs the file name associated with the inputed file descriptor. Upon the `libc.so's` return, the file name compares to the target filepath, and the `dumpstack()` method executes.

```
write_hook = Interceptor.attach(Module.findExportByName('libc.so',
    'write'),{
onEnter: function (args) {
    try{
    /*Write_filename is set to the associated file name in the
    file_descriptor array.*/
    this._write_fileName = file_descriptor_array[args[0].toInt32()]
    }
    catch{throw new Error("Invalid file_descriptor array reference");}
;
},
onLeave: function (retval) {
    if (retval.toInt32() > 0) { //Ensures valid return value
        //If file name equal to target file name, dump the stack
        if (typeof this._write_fileName !== 'undefined') {
            if (this._write_fileName.includes(target_filename) !== -1){
                console.log('\nFILENAME: ' + this._write_fileName);
                dumpStack();
                open_hook.detach();
                write_hook.detach();
    }}}
});
```

Figure 17: Javascript hook of `libc.so write()`.

As shown in Figure 18, the `dumpstack()` method is called from the write hook. `dumpstack()` grabs the correct virtual machine and creates a new environment in order to access and print the methods on the stack. It then passes in a thread from that environment into the `DebugStackVisitor` method.

```
function dumpStack() {
    if (Java.available){
        const vm = Java.vm;
        const env = vm.getEnv();
        Java.perform(function (){
            withRunnableArtThread(vm, env, thread => {
                const visitor = new DebugStackVisitor(thread);
                console.log('>>> walkStack');
                visitor.walkStack(true);
                console.log('<<< walkStack');
            });
        });
    }
    else
        console.log("Java NOT AVAILABLE");
}
```

Figure 18: Dump methods on the stack.

The `DebugStackVisitor()` method, found in Figure 19, takes in a thread and outputs method names and frame info on every method on the stack. As we discussed in section 3.2.6, the `ArtStackVisitor` class utilizes Native Android APIs to display information on the memory stack.

```
class DebugStackVisitor extends ArtStackVisitor {
    constructor(thread) {
        super(thread,
            getApi()['art::Thread::GetLongJumpContext'](thread),
            'include-inlined-frames');
    }
    visitFrame() {
        const location = this.describeLocation();
        console.log(`\t${location}`)
        const method = this.getMethod();
        if (method !== null){
            console.log(`\t\tArtMethod=${method.handle}`);
            const frameInfo = this.getCurrentQuickFrameInfo();
            if (frameInfo !== null) {
              console.log(`\t\tframeInfo=${JSON.stringify(frameInfo)}`);
            }
        }
        return true;
    }
}
```

Figure 19: Debugging Stack Visitor.

We clear the data and cache prior to hooking the `libc.so open()` to force the application to re-open the `data.mdb` file. Next, we execute `monkey -p com.yy.hiyo -c android.intent.category.LAUNCHER 1` to launch the default activity via the ADB interface. The Frida command in Figure 20 uses the frida-Java-playground environment to run user defined Javascript hooks on an application. The "-no pause" Frida argument is added to the Frida hook command to ensure that it hooks the libc open immediately after the application began running.

```
frida -U com.yy.hiyo -l _agent.js --enable-jit --no-pause
```

Figure 20: Frida command to inject hooks.

After executing the command in Figure 20, the application starts and the Javascript code in figures 15, 17, 18, and 19 is injected.

A simplified example output of a hooked write call can be found in Figure 21.

The value 2020 was written to the file and displayed in figure 21. In the case of non-readable type variables, Java prints the type of the variable. This requires more analysis to determine exactly what data is being written. In figure 21 there are three methods on the stack when the write call is made. Note that this displays only those methods on the stack and not necessarily every method called prior to the `libc.so write()`. The methods go from most recently called on the top to oldest on the bottom. In this example the io.value class's `store()` method is the last method called. The `bool` preceding the class name represents the return type of that method as a boolean operator. In some cases the walkstack output gives enough information to get a top-level view of how data is retrieved or manipulated prior to being written to a file. This may be enough for an examiners needs. In other cases, the data's format is still unclear and continued dynamic and static analysis is required.

In Figure 21, calls are made to `userinput()`, `add()`, and `multiply()` methods, and would likely infer that some values were inputted by the examiner and then immediately multiplied together. While the walkstack output can be useful, examiners follow up with static analysis to ensure that relevant method calls not on the stack are analyzed.

```
Written data: [2020]
>>> walkStack
[Top of Stack (Most Recently Called)]
    Visiting method 'bool io.value.store(int[])'
   Visiting method 'int[] io.value.add(int)'
   Visiting method 'int com.example.multiply(int[])'
   Visiting method 'int[] com.example.userinput()'
[Bottom of Stack]
<<< walkStack
```

Figure 21: Method Calls from Memory Stack

3.3.3 Native Method Hooking Step Three: Analyze Written Data

This subsection focuses on analyzing the data written to the file of interest. If the examiners are immediately able to identify the data's format, they would skip to step 5 of the DBIMAFIA methodology. Otherwise, the examiners should use the methods outlined in this subsection to attempt to determine the format of the bytes.

In many cases data is encapsulated within a Java object. If this is the case, examiners should use Java's `Object.println()` to obtain the ASCII representation of the class. Additionally, examiners run the `Object.getName()` method on the object to obtain the object's class name. If this is unsuccessful, examiners use online converters [87] [88] to convert the binary, decimal, or hex representation of the data to unicode, utf-8, utf-16, utf-32, and ASCII formats to identify readable characters.

The examiners now know which methods were called prior to file `libc.so` write, and have a better understanding of the data written to the file. If the examiners are

able to identify the storage format at this step, they should skip the class analysis section and begin synthesizing and validating their results.

3.4 Class Analysis

Step 4 of the DBIMAFIA methodology uses static analysis and dynamic binary instrumentation to analyze Java classes. The goal of this step is to provide the examiner with information to identify the purpose of every class that modifies user-data.

Examiners begin analysis on the method called prior to the file write. This method is at the top of the output of method calls output in Figure 21. After static analysis of `io.value.store()`, examiners use the code in Figure 22 to print the method's arguments, return value, and the method that called the hooked method.

The code in lines 8 through 35 of Figure 22 is run in place of the original method. To ensure the application runs properly, the original method is called on line 14 of Figure 22 and the result is saved and returned on line 30. Line 38 and 39 of Figure 22 defines the class and method to be hooked. Line 40 defines the types of arguments passed into the method.

This ability to re-implement any method allows the examiner access to any data that the original method would also have access to. In the case of the code in Figure 22, the examiner uses this to print the values found on the stack. However, Frida is not limited to reading values on the stack, Frida can read and modify data at the same permission level of the existing method call.

The examiner uses the calling method output to trace the method calls until they find the code responsible for manipulating or creating the user data. The code in Figure 22 gives the examiner the context to understand the purpose of each Java method, attribute and class.

```
function hook(obj, options) {
    var Exception = Java.use('Java.lang.Exception');
    var func = options['function'] !== undefined ? options['function'] :
        '$init';
    var args = options['arguments'] !== undefined ? options['arguments'] :
        [];
    var debug = options['debug'] !== undefined ? options['debug'] : false;
    var callOriginal = options['callOriginal'] !== undefined ?
       options['callOriginal'] : true;
    var callback = options['callback'] = options['callback'];
    try {
        Java.use(obj)[func].overload.apply(null, args).implementation =
           function() {
            var args = [].slice.call(arguments);
            var result = null;
            // Call Origin Function If True
            if (callOriginal) {
                result = this[func].apply(this, args, self);
            }
            if (callback) { // Call Callback If Exist
                result = callback(result, args);
            }
            // Debug Log
            if (debug) {
                var calledFrom =
                   Exception.$new().getStackTrace().toString().split(',')[1];
                var message = JSON.stringify({
                    arguments: args,
                    result: result,
                    calledFrom: calledFrom
                });
                console.log(obj + "." + func + "[\"Debug\"] => " + message);
            }
            // Return Result
            return result;
        };
    } catch (err) {// Error Log
        console.log(obj + "." + func + "[\"Error\"] => " + err);
    }
}
// Example Usage
Java.perform(function() {
    hook("com.example.app", {
        function: "a",
        arguments: ['Java.lang.CharSequence'],
        debug: true,
        callOriginal: true,
        callback: function(originalResult, args, self) {
            console.log("Args: " + args);
            console.log("Result: " + originalResult);
            return originalResult;
        }
    });
```

Figure 22: Method Hook.

3.5 Synthesis

The fifth step in the DBIMAFIA methodology synthesizes the findings from the class analysis phase to come to conclusions about how the application stores and manipulates user data. Illustrating individual method, attribute and class results in tables helps the examiner portray the results in an easy to read format. The examiner also ensures that findings from each class are synthesized to articulate how the application stores the user data.

3.6 Validation

The sixth and final step uses a byte debugger to determine the validity of conclusions made in step five.

Examiners use the smalidea plugin [48] with Android Studio to step through the `.DEX` files unpacked in Section 3.2.4. This does not require source code or repacking of the app's APK. Examiners set break points inside relevant methods identified in the class analysis and synthesis steps to validate:

1. The values held by Class attributes.

2. The functionality of identified methods and classes.

3. The control flow of the application's data serialization process.

Debuggers may not run on every application, due to the reasons outlined in Section 2.2.3 of the Background section of this thesis. In these instances the examiner has to rely on the static and Dynamic Binary Instrumentation (DBI) analysis techniques outlined in step four of the DBIMAFIA methodology.

3.7 Summary

In summary, the DBIMAFIA methodology covers device setup, the initial analysis of the APK and application files, native library hooking, class analysis, and synthesis and validation of results.

IV. Results and Analysis

4.1 Introduction

The results found by the Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) methodlogy is broken into a common format case study, and an unknown format case study. The common format case study section describes a situation where an examiner identifies a known storage format after initial analysis of the application of interest. The unkown format case study shows the results from the situation when the format is unkown and DBIMAFIA executes completely to determine the data types stored in the target file.

4.2 Common format case study

This section applies the DBIMAFIA methodology on the Samsung SmartThings mobile application that is found to store its data in an SQL Lite database [89].

4.2.1 Device Setup: SmartThings

This case study used a Google Pixel 1 running the Android 9.1.0 operating system, and the SmartThings Android application version 1.7.38-21. The phone was rooted using the process outlined in the methodology and was connected to a terminal running ADB version 28.0.2.

4.2.2 Initial Analysis: SmartThings

Initial Analysis Step One: Logical copy of App files and APK

Before executing the application, the examiner logically copied the unmodified

version of the `SmartThingsV_1-7-38-21.apk` and application related files over Android Debug Bridge (ADB).

Initial Analysis Step Two: Application Interaction

The application required the physical SmartThings hub to be connected to the internet and power before interaction could occur. Additionally, a multipurpose magnet sensor was attached to a door, and a Motion sensor was placed in an empty room. The examiners opened the application, connected the two sensors to the SmartThings application, and connected everything to WiFi. We verified that the devices were properly connected, and Modification Detective was run. The examiners interact with the physical sensors and modification Detective is stopped. This is repeated ten times to ensure consistent results. The only file that was consistently pulled back was the `CloudDb.db` file in the databases directory of the SmartThings application.

Initial Analysis Step Three: File Analysis

The linux `file` command returned the file type: `SQLite 3.x database`. The examiner then opened `CloudDb.db` with Sqllite, a common Structured Query Language (SQL) database viewer. Table 2 outlines the tables found inside the `CloudDb.db` database.

Table 2: SmartThings CloudDB.db Tables.

Table Name	Description
activity	Contains event log of every sensor input
android_metadata	Contains location and language data.
cloud_settings	Key value pairs of server and mobile device.
continuity_session	Records connection sessions with server.
devices	Information on all connected devices.
groups	Groups devices based on location.
locations	Table of all user defined locations.
plugin	Table of all device ids in order of connection.
robotcleaners	Holds current state of robot cleaner and location.
scenes	User configured logic based on sensor input.
scenes_action_value_type	History of actions taken by scene logic.

Modification Detective, a script found in Section 1.1 of the Appendix, revealed that when the smart sensors transmitted data, that they were written to the CloudDb.db file in the databases directory of the SmartThings application.

The activity log, found in Table 3, records the activity of every sensor connected to the SmartThings hub. Table 3 is an abbreviated version of the activity log table in the `CloudDb.db` file. The epoch column is a format that records the date and time that an event occurred. The `CloudDb` file also contained a device table that recorded the device id, location, state of sensor and epoch field of all IoT activity. Examiners found that the motion sensor collected and recorded motion sensor and temperature data.

Table 3: Abbreviated CloudDb.db activity log.

epoch	text	uiTimestamp
155803879500	contact of MP Sensor is: Open	1558038876206
155803879400	acceleration of MP Sensor is: Vibration detected	1558038876206
155803879300	motion of Motion Sensor is: Motion detected	1558038876206
155803879200	contact of MP Sensor is: Closed	1558038876206
155803879100	temperature of Motion Sensor is: 80°F	1558038876206

The examiners did not need to unpack or analyze the `SmartThingsV_1-7-38.apk`, because the data of interest was found in plaintext, the DBIMAFIA methodology indicates the examiner should skip to Step 5: Synthesis to summarize findings.

4.2.3 Synthesis: SmartThings

The SmartThings application was found to use the SQL Lite format to store its user data. Validation with the debugger was not required as both the file command, and the file extension noted the same file type.

4.2.4 Summary of Known Format Case Study

SmartThings was not alone in storing its application data in a known, unencrypted data format. Examiners applied the DBIMAFIA methodology to fifteen applications, fourteen of which used XML or SQL Lite formats to store their data. The outlier, Hago Games is discussed in detail in Section 4.3 of this chapter.

Table 4: Storage Formats of Analyzed Applications.

#	Application Name	Version	Description	Storage Format
1	August SmartLock	9.5.3	Lock	SQL Lite
2	SmartThings	1.7.38-21	IoT Hub	SQL Lite
3	Garmin Connect	4.22	Smart Watch	SQL Lite
4	Harmony	5.4.1	IoT Hub	SQL Lite
5	Wink	6.9.62.23006	IoT Hub	SQL Lite
6	Tile	2.51.0	Bluetooth Tracker	SQL Lite
7	Yale Connect	1.1.1	Lock	SQL Lite
8	Whats App	2.19.360	Messenger	SQL Lite
9	Instagram	123.0.0	Social Media	SQL Lite
10	Ludo King	4.8.0	Game	XML
11	Viber	11.7.05	Messenger	SQL Lite
12	Tinder	11.6.0	Dating	SQL Lite
13	TextNow	6.36.1	Messenger	SQL Lite
14	WPSOffice	12.0.1	Document Manager	XML
15	Hago Games	3.2.8	Game / Messenger	ObjectBox DB

The applications outlined in Table 4 further divide into two categories: IoT and non-IoT applications. The first 7 applications fall under the IoT category and were found to store information about physical sensors. This included data about an individual's GPS location, physical health, and security practices. The next 6 applications contained data about a user's messaging activity, photos, GPS location, and browser history.

As the SmartThings case study demonstrated in Table 3, users may not always know or understand all the data collected by their applications or Internet of Things

(IoT) devices. A motion sensor is not expected to collect temperature data. This may seem insignificant, but in the case of a classified environment, temperature data could indicate what is being stored in a building or room. The ability to determine the format of data stored by an application is vital to the security expert's ability to properly classify applications and devices.

4.3 Unknown format use cases

This section shows the results from applying DBIMAFIA to the Hago Games application [90] to determine how the application stores user data. This popular application has over 200 million downloads from the Google Play Store and supports voice and text-based communication with other players.

4.3.1 Device Setup: Hago Games

This case study used a Google Pixel 1 running the Android 9.1.0 operating system, and the Hago Games application version 3.2.8. Examiners rooted the device using the process outlined in Step One of methodology, and connectred it to a terminal running ADB version 28.0.2. We then installed Frida version 12.6.11 on the device.

4.3.2 Initial Analysis: Hago Games

Initial Analysis Step One: Logical copy of App files and APK

Before the application executes, an unmodified version of the Android application package (APK) and application related files were logically copied using the ADB `pull` command.

Initial Analysis Step Two: Application Interaction

After making a logical copy of the application and all of its files, Modification Detective was run. With the script running, we opened the Hago games application

and sent messages to other players and Modification Detective pulled back modified files. We repeated this process 10 times to ensure consistency and to identify the file saving the application's message data. Upon visual inspection of the pulled files, examiners found the `data.mdb` file that stored Hago Games' user data.

Initial Analysis Step Three: File Analysis

We used ADB to pull another copy of the application's file directory. Running a `diff` between the original `\files\db_ 12885822986` directory and the same directory after interaction, identified that the `data.mdb` file in the `\files\db_ 12885822986` directory was created after application login.

The `.mdb` extension indicates the file is a Microsoft database file [91], but initial attempts to open the file with Microsoft Access [92] resulted in an unrecognized database format error.

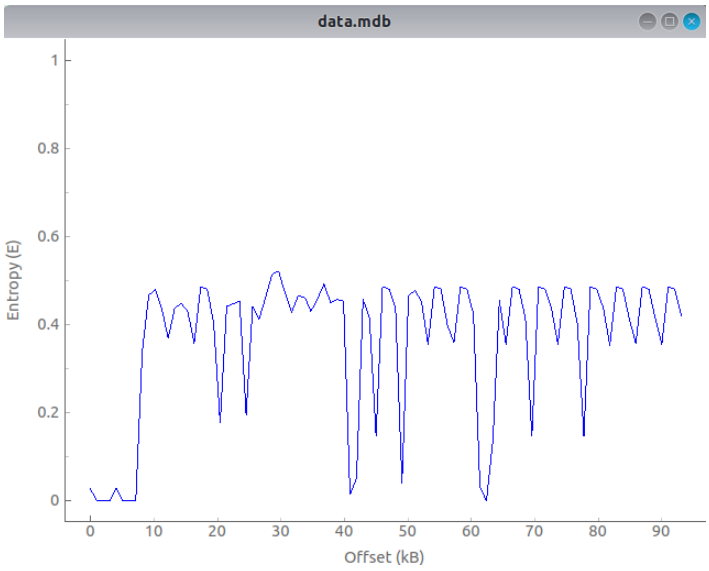

Figure 23: Entropy graph of data.mdb.

Running the Linux `file` command on the `data.mdb` file returned "data", indicating its format is not of a known file type. The Linux `binwalk -E` entropy command displayed the graph shown in Figure 23. Encrypted files typically have fairly consis-

tent entropy throughout; the constant change in entropy, indicated \files\db_12885822986\data.mdb was not encrypted.

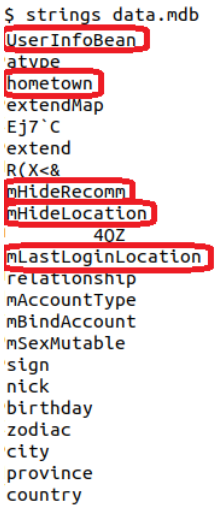

Figure 24: Strings in \files\db_12885822986\data.mdb.

As Figure 24 illustrates, the Linux strings command returned a series of decipherable American Standard Code for Information Interchange (ASCII) strings.

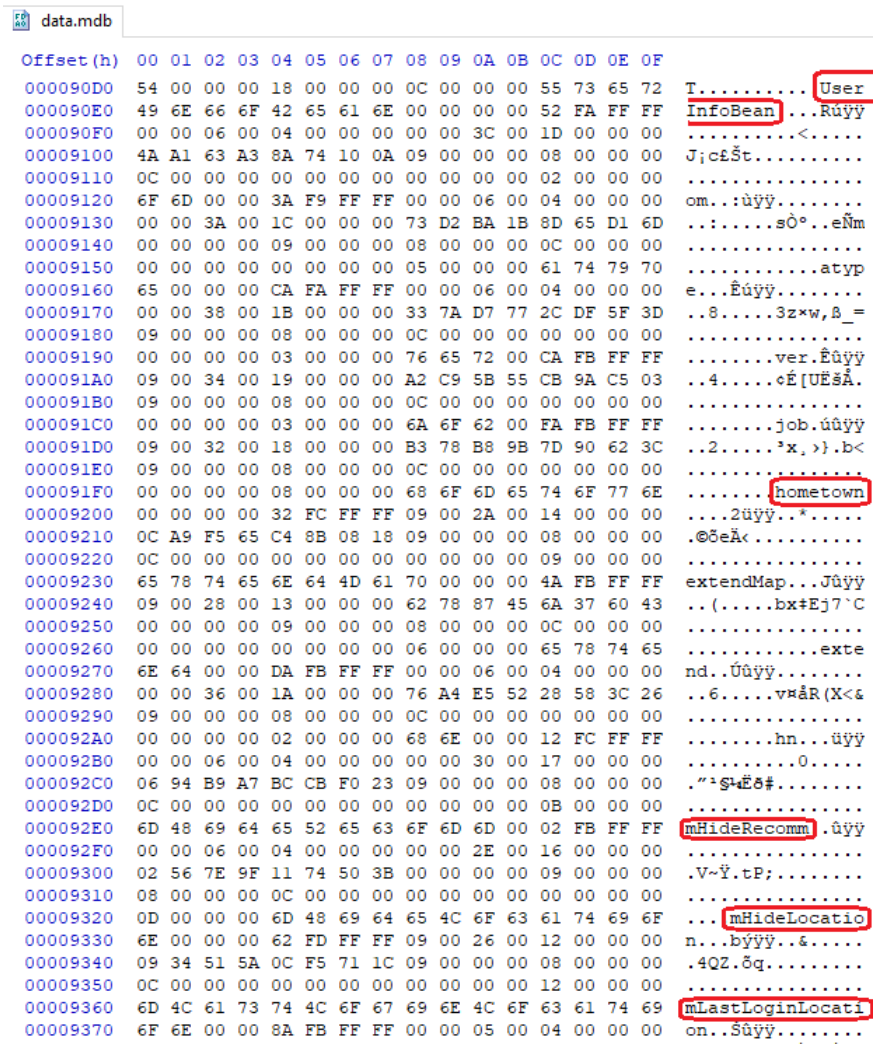

Figure 25: Hex representation of data.mdb.

We then opened `\files\db_12885822986\data.mdb` in HXD, a hex editor [93]. The examiners found the same strings output outlined in red in Figure 24, inside the `\files\db_12885822986\data.mdb` file in Figure 25.

Initial Analysis Step Four: Unpack and Decompile APK

The examiners opened the `HagoGamesV_3-2-8.APK` in JEB, and the application was unpacked and disassembled, exposing the Smali code of the application. The examiner now had direct access to the `manifest.xml` file, libraries, and the application resource files.

Initial Analysis Step Five: Identify Imported Libraries and Architecture

We analyzed the first line of the application's `manifest.xml` file to determine that Hago Games was using Android's software development kit version 28. We then inspected the application's imported libraries and found the `libobjectbox-jni` shared library file, a database library for mobile and IoT devices. Additionally, we found that the Hago application was using the ARM64v8 architecture. With a possible lead on a target file and related shared library, research now shifted to finding the code responsible for formatting the data written to the `\files\db_12885822986\data.mdb` file.

4.3.3 Native Method Hooking: Hago Games

Examiners next hook the `libc.so open()` and `write()` methods interacting with the `\files\db_12885822986\data.mdb` file. This step gave the examiner insight on where to begin class analysis.

Native Method Hooking Step One: Hook Open Methods and Obtain File Descriptor

We cleared the Hago Games data and cache prior to hooking the `libc.so open()` to force the application to re-open the `\files\db_12885822986\data.mdb` file. Next, we executed `monkey -p com.yy.hiyo -c android.intent.category.LAUNCHER 1` to launch the Hago Games default activity via the ADB interface. The Frida command in Figure 20 uses the frida-Java-playground environment to run user defined Javascript hooks on an application.

The code in Figure 15 reads the file path into the value variable and compares it with the predefined target filename (/data/user/0/ com.yy.hiyo/files/db_12885822986/) to determine if it should be saved in the file_descriptor array.

Native Method Hooking Step Two: Hook Write Methods, Dump Stack and Capture Saved Data

Once the examiner has the file descriptor associated with \files\db_12885822986 \data.mdb, the libc.so write() hook can be used to dump the stack.

```
[Google Pixel::com.yy.hiyo]->
Dump: /data/data/com.yy.hiyo/files/db_12885822986/data.mdb
>>> walkStack
[Top of Stack (Most Recently Called)]
   Visiting method 'long
      io.objectbox.BoxStore.nativeCreate(Java.lang.String, long, int,
      byte[])' at dex PC 0xffffffff (native PC 0x0)
   upcall
   Visiting method 'void
      io.objectbox.BoxStore.<init>(io.objectbox.b)' at dex PC 0x0066
   Visiting method 'io.objectbox.BoxStore io.objectbox.b.a()' at dex
      PC 0x001b
   Visiting method 'void com.yy.appbase.d.b.a(long)' at dex PC 0x003b
   Visiting method 'void com.yy.appbase.d.b.e()' at dex PC 0x000f
   Visiting method 'void com.yy.appbase.d.b.loginIn()' at dex PC
      0x0000
[Bottom of Stack]
<<< walkstack
```

Figure 26: Methods on the stack [1].

The examiner then logged into the Hago Games application, triggering a write to \files\db_12885822986\data.mdb, and the call stack in Figure 26 printed to the console. As Figure 26 indicates that io.objectbox.BoxStore.nativeCreate() is the last method called before the data is written to \files\db_12885822986\data.mdb.

The contents of arguments passed into io.objectbox.BoxStore.nativeCreate() were printed to the console. The first argument is the directory that stores the data.mdb file that is written to. Upon visual inspection, argument four appears to be an array of integers ranging from -128 to 127, that stores the data written to \files\db_12885822986\data.mdb.

[1] Note that frame information was removed for formatting purposes.

```
[Google Pixel::com.yy.hiyo]->
Method: io.objectbox.BoxStore.nativeCreate(Java.lang.String, long,
   int, byte[])
Arg1: /data/data/com.yy.hiyo/files/db_12885822986
Arg2: 1048576
Arg3: 0g
Arg4: [28,0,0,0,0,0,0,0,20,0,80,0,72,0,76,0,60,0,56,0,36,0,20,0,0,0,4,
0,20,0,0,,0,0,0,0,0,0,0,0,0,34,0,0,0,0,0,0,0,110,-119,-57,79 ...]
```

Figure 27: Debugging Stack Visitor[2].

Native Method Hooking Step Three: Analysis of Written Bytes

Examiners used the unicode converter found on branah.com [87] to convert the decimal array in argument 4 to unicode. Figure 28 shows that the bean names and variables found in `\files\db_12885822986\data.mdb` were also found in the byte array passed into `io.objectbox.BoxStore.nativeCreate()`. However, like the `data.mdb` file, all of the data in between the outlined strings is undecipherable. The non-ASCII characters indicate that this byte array contains non-unicode characters. We now had a basic level understanding of the bytes written to the file and their possible relation to the data inside `\files\db_12885822986\data.mdb`.

[2]The byte array in argument 4 was too long that it was not practical to include all the data in this document.

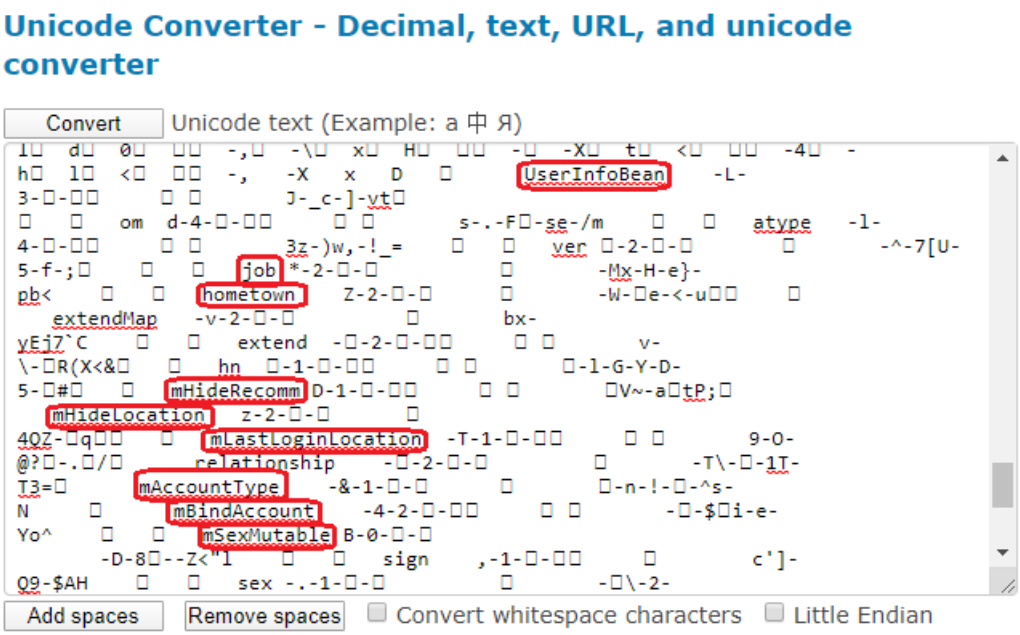

Figure 28: Unicode form of written bytes.

4.3.4 Class Analysis: Hago Games

In DBIMAFIA, step four, the examiner statically and dynamically analyzes methods involved with the formatting of the byte array written to `\files\db_12885822986\data.mdb`. In an ideal scenario, our original list of method calls would include every call involved with the formatting of the data. This was not the case for Hago Games. As this section discusses, there was a constant need to go back and forth between static analysis and Frida to track down the methods writing to the byte array.

We began static analysis with the native call made to `io.objectbox.BoxStore.nativeCreate()` that was executed from the constructor of `io.objectbox.BoxStore`. The byte array written to the database was saved inside the `io.objectbox.b.a` attribute.

After associating the byte array in argument four of Figure 27 with `io.objectbox.b`, we began dynamically analyzing the class and its attributes. Frida hooks on every

method of the class allowed examiners to dump the arguments passed into and returned from each method of the `io.objectbox.b` class. This combined with static analysis of the class, resulted in tables6 and 5 that explain the attributes and methods of the `io.objectbox.b` class.

Table 6: Methods of `io.objectbox.b BoxStore class`.

BoxStore Class		
Method	Return Type	Description
<init>(io.boxstore.b)	void	Runs BoxStoreBuilder
a(File)	String	Checks and returns file path
a(Class)	String	Return DB name
a(Transaction, int[])	void	Commit transactions
b(Class)	int	Return entity type Id
b(int)	String	Start Objectbox Browser
close()	void	Close boxstore
d(Class)	Box	Returns a box of given class
...		

Table 5: Attributes of `io.objectbox.b BoxStore class`.

Attribute	**Type**	**Description**
a	byte[]	model
b	file	directory
c	long	max size in KB
d	int	debug Flags
e	bool	debug Relations
f	int	max # readers
g	int	query attempts
h	TXCallback	helps sync data across devices
i	list	list of beans
j	file	directory
k	string	name
l	bool	
m	factory <inputStream>	input stream

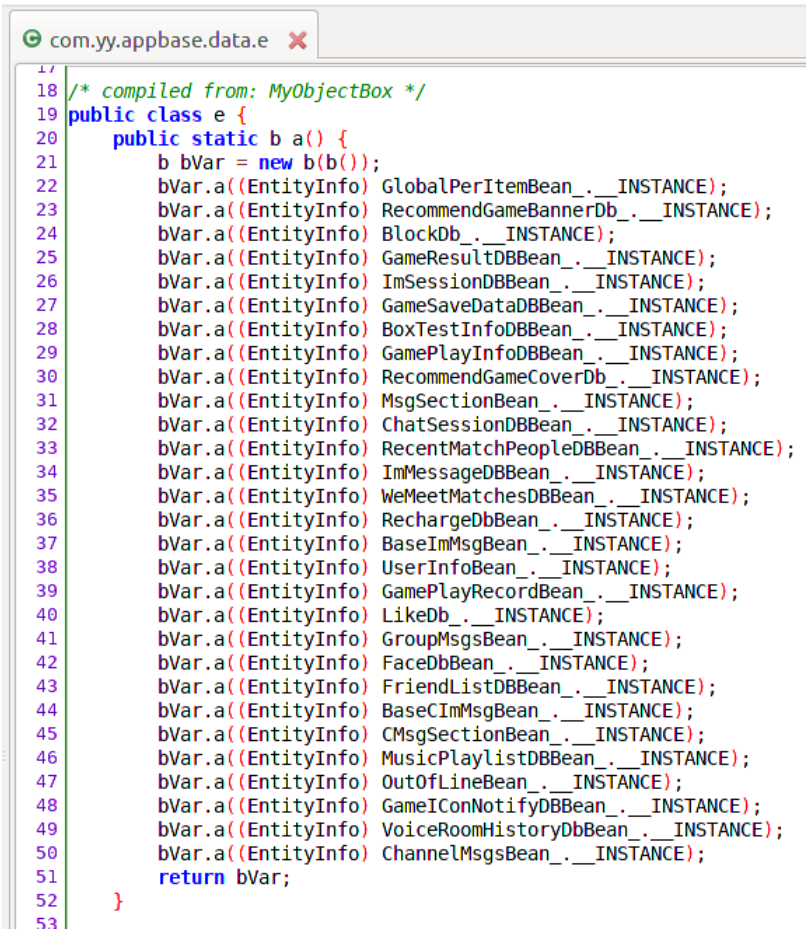

Figure 29: `com.yy.appbase.data.e.a()` method.

Examiners used Frida to hook `io.objectbox.b.a()` and print each individual bean object passed into it as an argument. This gave examiners a full list of every bean contained inside the greater `io.objectbox.b` object. A string search of the APK's code found references to the same list of beans in `com.yy.appbase.data.e`. Figure 29, line 21 shows the `io.objectbox.b` class being created. Lines 22 through 50 of Figure 29 reference 29 separate bean model classes to define the properties and instantiate each bean.

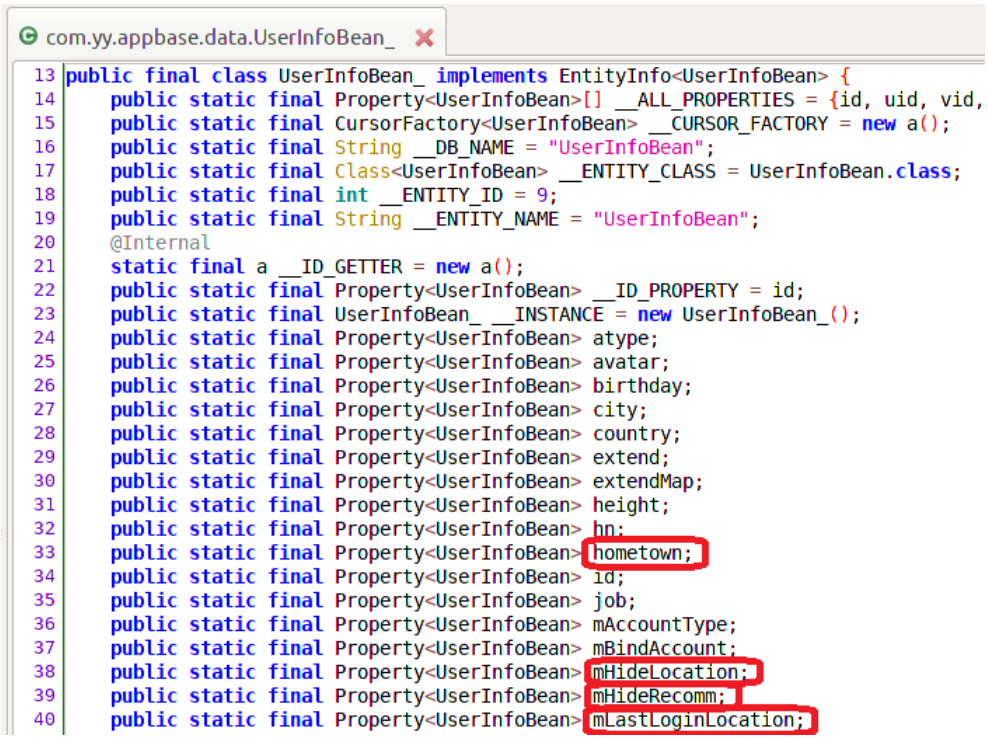

Figure 30: `com.yy.appbase.data.UserInfoBean objectBox.model class`.

Figure 30 contains the `UserInfoBean` model class, one of the 29 bean model classes identified by static analysis of `com.yy.appbase.data.e`. Notice the same strings were found in the `\files\db_12885822986\data.mdb` in Figure 25, and the byte array in Figure 28.

com.yy.appbase.data.e

```
57    private static byte[] b() {
58        d dVar = new d();
59        dVar.a(30, 59406390482908514801);
60        dVar.b(34, 1946943942591744366L);
61        dVar.c(0, 0);

330       a a17 = dVar.a("UserInfoBean");
331       a17.a(9, 3346480166988649961).b(29, 725207678802174282L);
332       a17.a(1);
333       a17.a(FacebookAdapter.KEY_ID, 6).a(1, 2173059337117973548L).a(5);
334       a17.a(ArgoProfileKey.ARGO_PROFILE_KEY_UID, 6).a(2, 449594409006371998L).a(12)
335       a17.a("vid", 6).a(21, 863954130636408054 3L).a(4);
336       a17.a("avatar", 9).a(3, 72984778471480816 62L);
337       a17.a("weight", 5).a(4, 3799998952798844424L).a(4);
338       a17.a("height", 5).a(5, 6805246134246711032L).a(4);
339       a17.a(u.COUNTRY, 9).a(6, 22375149106045243 82L);
340       a17.a("province", 9).a(7, 3172548514378104270L);
341       a17.a(u.CITY, 9).a(8, 19271647723937815 00L);
342       a17.a("zodiac", 6).a(9, 3571107432325023399L).a(4);
343       a17.a("birthday", 9).a(10, 64297563179496864 80L);
344       a17.a("nick", 9).a(11, 19501751089673248 99L);
345       a17.a(u.SEX, 5).a(12, 5206684784530368355L).a(4);
346       a17.a("sign", 9).a(13, 77918569897232160 60L);
347       a17.a("mSexMutable", 5).a(14, 6804841847905180923L).a(4);
348       a17.a("mBindAccount", 9).a(15, 9201544132250538523L);
349       a17.a("mAccountType", 9).a(16, 4410057940391255212L);
350       a17.a("relationship", 5).a(17, 3387219353400291641L).a(4);
351       a17.a("mLastLoginLocation", 9).a(18, 2049688738833970185L);
352       a17.a("mHideLocation", 6).a(22, 427404366541372365 0L).a(4);
353       a17.a("mHideRecomm", 6).a(23, 2589793796866282502L).a(4);
354       a17.a("hn", 6).a(26, 2755174002256618614L).a(4);
355       a17.a("extend", 9).a(19, 4854941327877961826L);
356       a17.a("extendMap", 9).a(20, 1731787732364405004L);
357       a17.a("hometown", 9).a(24, 4351199059151190195L);
358       a17.a("job", 9).a(25, 271793550626900386L);
359       a17.a(ServerTB.VER, 6).a(27, 44224987411832817 15L).a(4);
360       a17.a("atype", 6).a(28, 79132176769966946 43L).a(4);
361       a17.a("om", 6).a(29, 725207678802174282L).a(4);
362       a17.b();

503       return dVar.a();
504   }
```

Figure 31: `com.yy.appbase.data.e.b()` method.

We inspected the `com.yy.appbase.data.e.b()` method in Figure 31, and found that line 58 creates an `io.objectbox.d` object and then each of the 29 individual beans are added to the object in lines 62 through 502. Line 330 creates the new userInfoBean entity within the `io.objectbox.d` object. Lines 331 through 361 define its properties, and line 362 makes a call to the `io.object- box.d.b()` method to finalize the `UserInfoBean`. Examiners now used Frida hooks to print the arguments and return values of every method of `io.objectbox.d` to determine the purpose of

each method and attribute. Table 8 and Table 7 summarize these findings.

Table 7: Attributes of `io.objectbox.d`.

Attribute	Type	Description
a	FlatBufferBuilder	FlatBufferBuilder
b	List<Integer>	entity offsets
c	long	version #
d	int	last entity ID
e	long	last entity UID
f	int	last index ID
g	long	last index UID
h	int	last relation ID
i	long	last relation UID

Table 8: Methods of `io.objectbox.d`.

io.objectbox.d Class		
Method	Return Type	Description
<init>(io.objectbox.d)	void	
a(int, long)	io.objectbox.d.a	builds id property
a(list<int>)	int	creates an array of offsets and serializes as flatbuffer (integer)
a(int)	io.objectbox.d.a	builds flags
b()	io.objectbox.d	finalizes entity
b(int, long)	io.objectbox.d.a	builds index id property
c()	void	checks if more properties to add
...		

We hooked the `io.objectbox.d.a()` method to trace the call stack, and variable values throughout the method. Line 503 of Figure 31 calls the `io.objectbox.d.a()` method found in Figure 32. Figure 32, line 242 and 243 grab the name and object offsets for each bean and lines 244 and 247 add the offsets to the Google Flat Buffer Builder. Finally, on line 258, the flat buffer builder (fbb) is returned as a byte array into the a attribute of `io.objectbox.b`.

```
io.objectbox.d
240     public byte[] a() {
241         int a2 = this.a.a((CharSequence) "default");
242         int a3 = a(this.b);
243         io.objectbox.a.b.a(this.a);
244         io.objectbox.a.b.a(this.a, a2);
245         io.objectbox.a.b.a(this.a, 2);
246         io.objectbox.a.b.b(this.a, 1);
247         io.objectbox.a.b.b(this.a, a3);
248         if (this.d != null) {
249             io.objectbox.a.b.c(this.a, io.objectbox.a.a.a(this.a, (long) this.d.intValue(), this.e.longValue()));
250         }
251         if (this.f != null) {
252             io.objectbox.a.b.d(this.a, io.objectbox.a.a.a(this.a, (long) this.f.intValue(), this.g.longValue()));
253         }
254         if (this.h != null) {
255             io.objectbox.a.b.e(this.a, io.objectbox.a.a.a(this.a, (long) this.h.intValue(), this.i.longValue()));
256         }
257         this.a.h(io.objectbox.a.b.b(this.a));
258         return this.a.f();
259     }
260 }
```

Figure 32: `io.objectbox.d.a()` method.

We used Frida to print the stack's contents right before the byte array was returned on line 258 of Figure 32. Figure 33 is a call graph representation Frida's output.

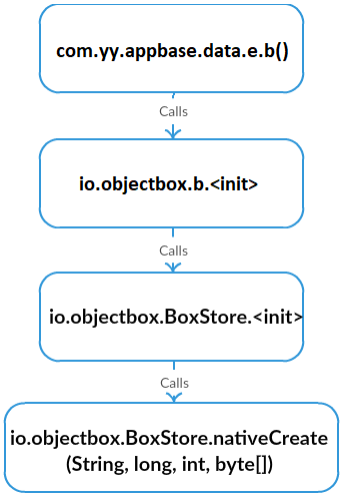

Figure 33: Call trace of byte array manipulations.

4.3.5 Synthesis: Hago Games

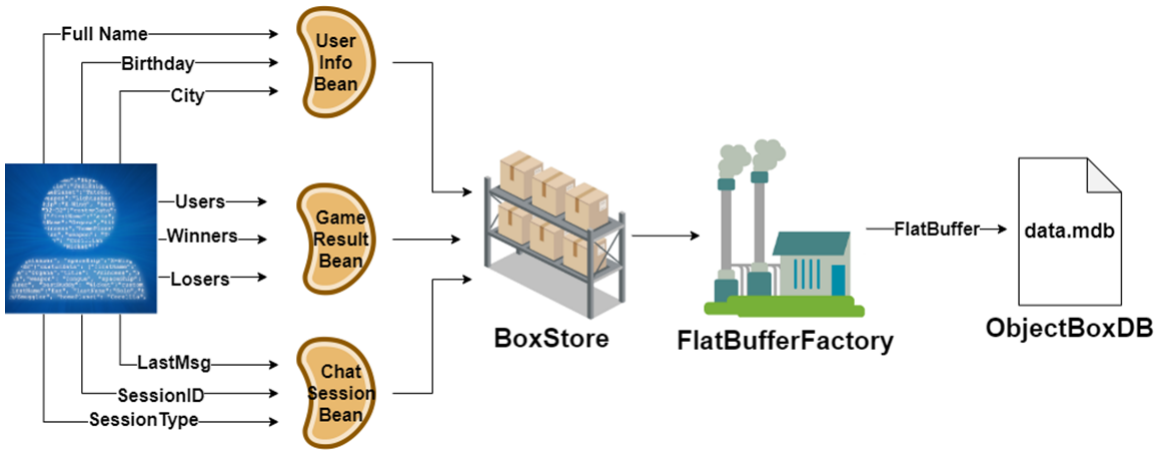

Figure 34: Hago Games Data Serialization Process

The creation of the `\files\db_12885822986\data.mdb` is shown in Figure 34. This process is further explained in three main steps:

1. `com.yy.appbase.data.e.b()`
   (a) Creates model creator (`io.objectbox.d`).
   (b) Adds beans and their attributes to the model creator.
   (c) Serializes beans as fbb.
   (d) Returns fbb.
2. `io.objectbox.b.<init>`
   (a) Calls `com.yy.appbase.data.e.a()` to add bean.
   (b) Saves beans serialized as fbb byte array into the a attribute of the BoxStore Creator (`io.objectbox.b`).
3. `io.objectbox.BoxStore`

(a) Passes fbb as byte array to `io.objectbox.BoxStore.nativeCreate()` to create `\files\db_12885822986\data.mdb`.

The format of the byte array is modeled in Table 9. The beans and their attributes are all stored inside the array passed into `io.objectbox.BoxStore.nativeCreate()`. The `io.objectbox.BoxStore` class itself is representative of `\files\db_12885822986 \data.mdb`.

These findings on how Hago Games uses ObjectBox library to store data in custom file formats can be applied more broadly to any device or application using the ObjectBox library. ObjectBox advertises itself as an edge database for IoT and mobile applications. Since its release in 2016 ObjectBox has steadily grown in popularity. The Objectbox library is not just popular in mobile applications, the library is used in railway systems and continues to be used in various IoT and SCADA systems [94].

The Hago Games stored message history, location information, and personal data on users. Our research found that any conversation with another Hago user resulted in a `userInfoBean` being stored for that user. The `userInfoBean` was created immediately after a message was sent and did not require the user to reply. This `userInfoBean` contained location and birthday information that is not visually displayed in the application. This meant that examiners could send any user a message and expose their full birthday and city. Users are unknowingly exposing their birth dates and cities by using the Hago Games application. Examiners' use of `DBIMAFIA` gave them the ability to understand how Hago Games stored user data, which led to the discovery of unnecessary personally identifiable information (PII) leakage.

4.3.6 Validation

We used Android Studio and the smalidea plugin [48] to bytecode debug the application and validate our findings. We traced the byte array through the `io.objectbox.`

Table 9: Format of Byte Array.

	MyObjectBox Format					
BeanName	GlobalPerItemBean	RecommendGameBannerDb	BlockDb	GameResultDBBean	ImSessionDBBean	GameSaveDataDBBean
Attributes	long id String Ac String time ...	Int displayTimes String gameId String videoUrl ...	long id long timestamp long uid	String losers String users String winners ...	Int bindType int chatType String contentType ...	String context long id String key ...
BeanName	BoxTestInfoDBBean	GamePlayInfoDBBean	RecommendGameCoverDb	MsgSectionBean	ChatSessionDBBean	RecentMatchPeopleDBBean
Attributes	String avatar String birthday String city ...	long endTs String gameId int gameMode ...	String gameId long id String imgUrl ...	String color String content String ext ...	String lastmsg String sessionID Int sessionType ...	String avatar String gameIcon String gameId ...
BeanName	ImMessageDBBean	WeMeetMatchesDBBean	RechargeDbBean	BaseImMsgBean	UserInfoBean	GamePlayRecordBean
Attributes	transient BoxStore int bindType int chatType ...	String avatar long id bool isOnline ...	Int chargeConfigId long diamond String gameId ...	String avatar String cid String cname ...	String avatar String birthday String city ...	String gameId int gameMode long gamingTs ...
BeanName	LikeDb	GroupMsgsBean	FaceDbBean	FriendListDBBean	BaseCImMsgBean	CMsgSectionBean
Attributes	long id long uid	transient BoxStore String groupId long id ...	bool available bool cold String faceId ...	String extend String extendMap String extendTwo ...	transient BoxStore String avatar String cid ...	String color String content String ext ...
BeanName	MusicPlaylistDBBean	OutOfLineBean	GameIConNotifyDBBean	VoiceRoomHistoryDbBean	ChannelMsgsBean	
Attributes	long addTimestamps long id bool isFileExist ...	long id int invalidType bool isNeverShow ...	String content long endTs String gameId ...	long timeStamp String roomId long id ...	transient BoxStore String groupId long id ...	

`BoxStore`, `io.objectbox.b`, `com.yy.appbase.data.e`, and `io.objectbox.d` classes. We set break points on all of these classes to validate the purpose of their methods and attributes. We stepped through the calls to various `io.objectbox.d` method calls inside the `com.yy.appbase.data.e.b()` method and watched the fbb of each bean being created. Finally, we traced the byte array from the `io.objectbox.d.a()` method to the `io.objectbox.BoxStore` class and viewed the contents of both the fbb byte array passed into the `nativeCreate()` method and the `io.objectbox.b` object holding the traits of all the beans and their offsets. The examiners found that the debugger's findings validated DBIMAFIA's results.

4.4 Summary

In summary, the application of the DBIMAFIA methodology was applied to the 15 Android applications found in Table 4. In Section 4.2, we found that 14 of our 15 applications used common formats to store user data. These applications stored user's GPS locations, messaging information, and private data.

In Section 4.3, we discovered that Hago Games used the ObjectBox shared library to format and save data. The ObjectBox library used fbbs, and Java beans to serialize the user data into `\files\db_12885822986\data.mdb`. Upon inspection of the data stored by the Hago Games app, it was discovered that user's birthdays and cities were being unnecessarily exposed to other users. Furthermore, these findings on how Hago Games uses ObjectBox library to store data in custom file formats can be applied more broadly to any mobile, IoT, or SCADA device or application using the ObjectBox library.

V. Conclusions

This thesis demonstrates that dynamic instrumentation tools combined with static analysis tools can effectively determine the format of data stored by Android applications.

This paper presented a new approach to reverse engineering unknown file formats of Android Applications. Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis (DBIMAFIA) combines static analysis with the use of dynamic instrumentation tools to provide reverse engineers with a new way to approach format analysis of Android application data. Mobile application developers commonly obfuscate applications and add code to intentionally stop reverse engineers from debugging their code. Use of dynamic instrumentation tools, like Frida, avoids the need to modify the application's code or properly decompile an application's code. The result is a process that circumvents the common anti-debug techniques implemented by the developers.

With Samsung SmartThings and the thirteen other applications outlined in Table 4, DBIMAFIA successfully Identified the format of data stored. Examiners found that these applications used common formats to store their data and applied DBIMAFIA to quickly identify the location and format of relevent data. Examiners discovered that applications one through fourteen in table 4 stored user's GPS locations, messaging information, and physical security practices in SQL Lite or XML formats.

With Hago Games, DBIMAFIA successfully identified the format of user data stored. This application required the use of the Dynamic Binary Instrumentation (DBI) tool, Frida, to trace the written byte array through its manipulations. Frida hooks of Java methods were used to print the contents of the byte array and associated objects to reveal the purpose of related classes, methods, and attributes. Examiners

combined the use of Frida with static analysis to determine the objectBox format of the Hago Games application. Examiners used their understanding of the objectBox format to discover that full names, birthdays, and cities were being unnecessarily exposed to other users.

5.1 Impact

The variety of Internet of Things (IoT) devices, mobile devices, and firmware versions, provide a challenge for security experts. The security approach of today's desktop computers may not be practical as the cyber battle space continues to evolve. Discovering ways to dynamically analyze mobile and IoT data is one of many steps that must be taken as the Department of Defense (DoD) becomes more reliant on a wider variety of electronic devices. This research explored how dynamic instrumentation tools can be used to aid an examiner analyzing mobile application data. The DBIMAFIA methodology can be applied to aid forensic investigations, and the classification of mobile devices. More directly, Hago users can be made aware of the leakage of their birth dates and locations to other users. Furthermore, these findings on how Hago Games uses ObjectBox library to store data in custom file formats can be applied more broadly to any mobile, IoT, or SCADA device or application using the ObjectBox library.

5.2 Future Work

The DoD recognizes that large collections of unclassified information poses a threat to the United States. IoT and mobile device data is no different. Small amounts of IoT and mobile data exposure might seem like a limited threat to the DoD, but aggregated pools of mobile and IoT data could pose a threat to the security of this nation. Research into what conclusions can be made from large scale analysis of

aggregated IoT data would be invaluable to security experts.

The DBIMAFIA methodology successfully identified the format of user data on an application that stored data in an unkown format. Applying the DBIMAFIA methodology to determine the format of an application that encrypts its user data would be an interesting case study.

This research found offsets within the code of the Hago games application. Future research into using these offsets to design a parser for objectbox data files would be of great value to investigators who want to quickly access the user data inside of any mobile application or IoT or SCADA device using the ObjectBox library to save data.

Appendix A. Appendix

1.1 Modification Detective Source Code

```
import sys, subprocess, time

# Creates a new file that is later used as a timestamp to find files changed since this file was
    created
output_path = ""
bashCommand = "adb shell \"su -c touch /sdcard/new_file" + "\""
print('\t[+] ' + bashCommand)
output_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))

# Delay for user to run application as desired
delay= 15;
for x in range(delay):
   print("Progress {:2.1%}".format(x / delay), end="\r")
   time.sleep(1)

# Hardcoded path to applications data/data folder
folder = "/data/data/com.amazon.dee.app*"

# compiles a list of all files changed in th eapplications data foldersince the creation of new_file
    (~ 30 seconds earlier)
bashCommand = "adb shell \"su -c find " + folder + " -type f -newer /sdcard/new_file >
    /sdcard/list_files.txt" + "\""
print('\t[+] ' + bashCommand)
output_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))

time.sleep(1)

# Pulls a list of files that have been changed back to host
bashCommand = "adb pull /sdcard/list_files.txt"
print('\t[+] ' + bashCommand)
output_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))
```

```
# Creates a temporary directory for changed files
bashCommand = "adb shell \"su -c mkdir /sdcard/temp" + "\""
print('\t[+] ' + bashCommand)
output_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))

# Reads in the filenames line by line into a string
fileName ="list_files.txt"
with open(fileName) as f:
  files = f.readlines()
files = [line.rstrip('\n') for line in open(fileName)]

# MAkes of copy of each changed file in the temp directory created earlier
for name in files:
   bashCommand = "adb shell \"su -c cp " + name + " /sdcard/temp" + "\""
   print('\t[+] ' + bashCommand)
   utput_byte = subprocess.check_output(['bash','-c', bashCommand])
   print("\t" + str(output_byte, 'utf-8'))

time.sleep(1)

# Pulls the folder of edited files back to host
bashCommand = "adb pull /sdcard/temp"
print('\t[+] ' + bashCommand)
utput_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))

time.sleep(1)

# Cleans up and removes all created folders and files
bashCommand = "adb shell \"su -c rm -r /sdcard/temp" + "\""
print('\t[+] ' + bashCommand)
output_byte = subprocess.check_output(['bash','-c', bashCommand])
print("\t" + str(output_byte, 'utf-8'))
```

Bibliography

1. Google Developers, "Android Application Fundamentals," 2019. [Online]. Available: https://developer.android.com/guide/components/fundamentals

2. A. Nazar, M. Seeger, and H. Baier, "Rooting Android – Extending the ADB by an Auto-Connecting WiFi-Accessible Service," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7161, no. October 2011, pp. 1–3, 2011. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84862142324&partnerID=tZOtx3y1

3. D. Barry and T. Stanienda, "Solving the Java object storage problem," *Computer*, vol. 31, no. 11, pp. 33–40, 1998.

4. M. Altarade, "The definitive guide to NoSql," 2016. [Online]. Available: https://www.toptal.com/database/the-definitive-guide-to-nosql-databases

5. Frida, "Dynamic Instrumentation Toolkit," 2019. [Online]. Available: https://www.frida.re/

6. TwoSix Labs, "Edge of the Art in Vulnerability Research DARPA CHESS Program," vol. 22203, no. December, 2019.

7. D. Sazonov, "Andriller - Android Forensic Tools," *Git Hub*, 2019. [Online]. Available: https://github.com/den4uk

8. D. Trump, "Executive Order on Securing the Information and Communications Technology and Services Supply Chain," 2019. [Online]. Available: https://www.whitehouse.gov/presidential-actions/executive-order-securing-information-communications-technology-services-supply-chain/

9. StatCounter, "Mobile Operating System Market Share Worldwide," 2019. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

10. United States Government, "Federal Government Mobile Applications Directory," 2019. [Online]. Available: https://www.usa.gov/mobile-apps#focusable

11. J. Marks, "DOD Opts for Android for Classified Tablets," 2017. [Online]. Available: https://www.nextgov.com/it-modernization/2017/08/dod-opts-android-classified-tablets/140069/

12. G. Seffers, "Maxwell-Gunter Air Force Base serves as the IoT example." 2018. [Online]. Available: https://www.afcea.org/content/air-force-extends-smart-base-pilot-program

13. T. Armerding, "The 18 biggest data breaches of the 21st century: Security practitioners weigh in on the 18 worst data breaches in recent memory." *Chief Security Office United States*, pp. 1–7, 2018. [Online]. Available: https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html

14. K. DiGrazia, "Cyber Insurance, Data Security, and Blockchain in the Wake of the Equifax Breach," *Journal of Business and Technology Law*, vol. 13, no. 2, pp. 255–277, 2018. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=edshol&AN=edshol.hein.journals.jobtela13.16&lang=de&site=eds-live

15. N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019.

16. AT&T Business, "Smart Base Pilot Program: IoT solutions for military instillations," 2018. [Online]. Available: https://www.business.att.com/content/dam/attbusiness/insights/casestudiesandpdfs/CS_Maxwell_AFB.pdf

17. A. Schiffer, "How a fish tank helped hack a casino," *The Washington Post*, pp. 205–220, 2017. [Online]. Available: https://www.washingtonpost.com/news/innovations/wp/2017/07/21/how-a-fish-tank-helped-hack-a-casino/

18. S. Keane, "Huawei ban: Full timeline as House bars US government from buying Chinese company's gear," 2019. [Online]. Available: https://www.cnet.com/news/huawei-ban-full-timeline-house-us-government-china-trump-ban-security-threat-mate-x/

19. K. Rosenblatt, "Army bans TikTok following guidance from the Pentagon," 2019. [Online]. Available: https://www.nbcnews.com/news/amp/ncna1109001

20. J. Sanders and D. Patterson, "Facebook data privacy scandal: A cheat sheet," 2019. [Online]. Available: https://www.techrepublic.com/article/facebook-data-privacy-scandal-a-cheat-sheet/

21. AppBrain, "Hago- Play with new friends: Games Statistics," 2019. [Online]. Available: https://www.appbrain.com/app/hago-play-with-new-friends/com.yy.hiyo

22. August Home Inc, "August Home," *Google Play Store*, vol. 9.5.3, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.august.luna&hl=en_US

23. L. Samsung Electronics Co., "Samsung SmartThings," *Google Play Store*, vol. 1.7.38-21, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.samsung.android.oneconnect&hl=en_US

24. Garmin, "Garmin Connect," *Google Play Store*, vol. 4.22, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.garmin.android.apps.connectmobile&hl=en_US

25. WhatsApp Inc., "WhatsApp Messenger," *Google Play Store*, vol. 2.19.360, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.whatsapp&hl=en_US

26. Instagram, "Instagram," *Google Play Store*, vol. 123.0.0, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.instagram.android&hl=en_US

27. Gametion Technologies Pvt Ltd, "Ludo King™," *Google Play Store*, vol. 4.8.0, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.ludo.king&hl=en_US

28. Viber Media S.à r.l., "Viber Messenger - Messages, Group Chats & Calls," *Google Play Store*, vol. 11.7.05, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.viber.voip&hl=en_US

29. Tinder, "Tinder," *Google Play Store*, vol. 11.6.0, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.tinder&hl=en_US

30. I. TextNow, "TextNow: Free Texting & Calling App," *Google Play Store*, vol. 6.36.1, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.enflick.android.TextNow&hl=en_US

31. Kingsoft Office Software Corporation Limited, "WPS Office - Word, Docs, PDF, Note, Slide & Sheet," *Google Play Store*, vol. 12.0.1, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=cn.wps.moffice_eng&hl=en_US

32. Logitech Europe S.A., "Harmony," *Google Play Store*, vol. 6.2.1, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.logitech.harmonyhub&hl=en_US

33. Wink App, "Wink - Smart Home," *Google Play Store*, vol. 6.9.62.230, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.quirky.android.wink.wink&hl=en_US

34. Tile Inc., "Tile," *Google Play Store*, vol. 2.51.0, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.thetileapp.tile&hl=en_US

35. ASSA ABLOY Americas International Logistic Co, "Yale Connect," *Google Play Store*, vol. 1.1.1, 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.assaabloy.yaleconnect&hl=en_US

36. S. Reti, G. D'Angelo, and A. Omicini, "Hooking Java methods and native functions to enhance Android applications security," 2016. [Online]. Available: https://amslaurea.unibo.it/12257/1/Brandolini_HookingJavaMethodsAndNativeFunctions.pdf

37. Oracle, "What is a Database," 2018. [Online]. Available: https://www.oracle.com/database/what-is-database.html

38. ——, "The Java EE 8 tutorial," 2017. [Online]. Available: https://javaee.github.io/tutorial/ejb-intro.html

39. E. Chikofsky and J. Cross, "1990 - RE and Design Recovery," no. January, pp. 13–17, 1990.

40. R. Dill, "Automating Mobile Device File Format Analysis," 2018. [Online]. Available: https://apps.dtic.mil/docs/citations/AD1063269

41. B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh, "Industrial Perspective on Static Analysis." *Software Engineering Journal*, no. March, pp. 69–75, 1995. [Online]. Available: http://www.ida.liu.se/~TDDC90/papers/industrial95.pdf

42. APKPure LLC, "APKPure Homepage," 2014. [Online]. Available: https://apkpure.com/

43. Illogical Robot LLC, "APK Mirror," 2014. [Online]. Available: https://www.apkmirror.com/faq/

44. KingoApp, "What Role does SU binary play in Android rooting?" p. 1, 2019. [Online]. Available: https://www.kingoapp.com/knowledge-base/what-is-su-binary.htm

45. J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," *Internet Engineering Task Force*, 1996. [Online]. Available: InternetEngineeringTaskForce

46. I. Pavlov, "7Zip Installation," 2019. [Online]. Available: https://www.7-zip.org/

47. Skylot, "jadx - Dex to Java decompiler," 2019. [Online]. Available: https://github.com/skylot/jadx

48. Jesus Freke, "Samli," 2019. [Online]. Available: https://github.com/JesusFreke/smali

49. B. Gruver, "smali Package Description," 2009. [Online]. Available: https://tools.kali.org/reverse-engineering/smali

50. PNF Software, "JEB," 2019. [Online]. Available: https://www.pnfsoftware.com/jeb2/manual/debugging/#availability

51. Hex-Rays, "Debugging Dalvik programs with IDA," 2014. [Online]. Available: https://www.hex-rays.com/products/ida/support/tutorials/debugging_dalvik.pdf

52. Google, "Android Studio," 2019. [Online]. Available: https://developer.android.com/studio/releases/

53. Check Point Software Technologies LTD., "Cuckoo-droid," 2015. [Online]. Available: https://github.com/idanr1986/cuckoo-droid

54. Joe Security, "Joe Sandbox," 2019. [Online]. Available: https://www.joesecurity.org/joe-sandbox-mobile

55. C. Tumbleson, "Apktool v2.2.0 Released," 2016. [Online]. Available: https://connortumbleson.com/2016/08/07/apktool-v2-2-0-released/

56. H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on android platform," *Journal of Supercomputing*, vol. 72, no. 1, pp. 232–246, 2016.

57. Guardsquare, "Protecting Android applications and SDKs against reverse engineering and hacking with Dexguard," 2019. [Online]. Available: https://www.guardsquare.com/en/products/dexguard

58. Licel Corporation, "What is DexProtector?" 2019. [Online]. Available: https://dexprotector.com/

59. E. Eilam, *Reversing: Secrets of Reverse Engineering.* Wiley Publishing Inc., 2005.

60. Microsoft, "Dumpbin Reference," 2016. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/dumpbin-reference?view=vs-2019

61. G. C. Kessler, "Are mobile device examinations practiced like 'forensics'?" *Digital Evidence and Electronic Signature Law Review*, vol. 12, no. 0, 2015.

62. M. Pollitt, "OSAC Technical Series 0002 A Framework for Harmonizing Forensic Science Practices and Digital / Multimedia Evidence OSAC Technical Series 0002 A Framework for Harmonizing Forensic Science Practices and Digital / Multimedia Evidence."

63. P. Stelfox, *Criminal investigation: An introduction to principles and practice*. Willan, 2013.

64. S. Bommisetty, R. Tamma, and H. Mahalik, *Practical Mobile Forensics*, Birmingham B3 2PB, UK., 2014.

65. M. Hassan and L. Pantaleon, "An investigation into the impact of rooting android device on user data integrity," *Proceedings - 2017 7th International Conference on Emerging Security Technologies, EST 2017*, pp. 32–37, 2017.

66. E. Casey and B. Schatz, *Conducting digital investigations*.

67. N. Bergman, J. Rouse, M. Stanfield, J. Scambray, S. Deshmukh, M. Price, S. Geethakumar, S. Matsumoto, and J. Steven, *Hacking Exposed Mobile: Security Secrets & Solutions*. McGraw Hill Professional, 2013.

68. Nation Security Agency, "GHIDRA: A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission," 2019. [Online]. Available: https://ghidra-sre.org/

69. Vector35, "Binary Ninja: A New Type of Reversing Platform," 2016. [Online]. Available: https://binary.ninja/

70. utds3 lab, "Multiverse: Static Binary Rewriter," 2019. [Online]. Available: https://github.com/utds3lab/multiverse

71. Gramma Tech, "DDISASM: Datalog Disassembly," 2019. [Online]. Available: https://github.com/GrammaTech/ddisasm

72. R. O'Callahan, "rr- reverse engineering tool," *Git Hub*, 2019. [Online]. Available: https://github.com/mozilla/rr

73. Mozilla Research, "How rr works," 2019. [Online]. Available: https://rr-project.org/

74. Lody, "Legend android," *Git Hub*, 2019. [Online]. Available: https://github.com/asLody/legend

75. C. Mulliner, "adbi - The Android Dynamic Binary Instrumentation Toolkit," *Git Hub*, 2015. [Online]. Available: https://github.com/crmulliner/adbi

76. R. Spolaor, E. D. Santo, and M. Conti, "DELTA: Data Extraction and Logging Tool for Android," *IEEE Transactions on Mobile Computing*, vol. 17, no. 6, pp. 1289–1302, 2018.

77. H. Falaki, R. Mahajan, and D. Estrin, "SystemSens: A tool for monitoring usage in smartphone research deployments," *proceedings 6th international Workshop MobiArch*, pp. 25–30, 2011. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1999916.1999923

78. J. Grover, "Android forensics: Automated data collection and reporting from a mobile device," *Proceedings of the Digital Forensic Research Conference, DFRWS 2013 USA*, vol. 10, pp. S12–S20, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2013.06.002

79. R. Guo, T. Zhu, Y. Wang, and X. Xu, "Mobilesens: A framework of behavior logger on Android mobile device," *IEEE*, no. 6th International conference, pp. 281–286.

80. A. Nandugudi and E. Al., "PhoneLab: A large programmable smartphone testbed," *Workshop Sensing Big Data Mining*, pp. 1–6, 2013.

81. C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: Measuring wireless networks and smartphone users in the field," *Special interest group on performance evaluationS*, 2011.

82. D. T. Wagner, A. Rice, and A. R. Beresford, "Device analyzer: Understanding smartphone usage," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, vol. 131, pp. 195–208, 2014.

83. Magisk, "Root Magisk without TWRP on Android P," 2018. [Online]. Available: https://forum.xda-developers.com/pixel-2-xl/how-to/guide-magisk-twrp-android-p-t3826994

84. O. A. Ravnås, "Art-internals probe," 2017. [Online]. Available: https://github.com/oleavr/art-internals/blob/070aebfc5e96c8ff5977875c790c75c87dc6ac55/probe.py#L188

85. N. Jones, "How to Use C's offsetof() Macro," 2004. [Online]. Available: https://barrgroup.com/embedded-systems/how-to/c-offsetof-macro

86. Linux Community, "Linux man pages," 2019. [Online]. Available: https://linux.die.net/man/2/

87. Branah, "Unicode Converter - Decimal, text, URL, and unicode converter," vol. 1.0, 2019. [Online]. Available: https://www.branah.com/unicode-converter

88. RapidTables, "RapidTables Number Conversion," 2019. [Online]. Available: https://www.rapidtables.com/convert/number/index.html

89. E. B. Bloomberg, Navigation Data Standard, "What Is SQLite?" 2020. [Online]. Available: https://www.sqlite.org/index.html

90. H. Games, "HAGO - Play with firends," 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.yy.hiyo&hl=en_US

91. FileInfo, "Whats is the .mdb file extension and how can I open it?" 2017. [Online]. Available: https://fileinfo.com/extension/mdb

92. TutorialsPoint, "Microsoft Access - Overview," 2015. [Online]. Available: https://www.tutorialspoint.com/ms_access/ms_access_overview.htm

93. Maël Hörz, "HxD - Freeware Hex Editor and Disk Editor," *mh-nexus*, vol. 2.3.0, 2019. [Online]. Available: https://mh-nexus.de/en/hxd/

94. Kapsch, "Industrial IoT (IIoT) edge solution for railway operators – a Kapsch ObjectBox Case Study," 2018. [Online]. Available: https://objectbox.io/iiot-edge-solution-railway-industry-kapsch-objectbox-case-study/

Acronyms

ADB Android Debug Bridge. 20, 38, 56

ADBI Android Dynamic Binary Instrumentation. 33

ADSS Automated Data Structure Slayer. 35

API Application Program Interface. 4, 9, 26

APK Android application package. vii, 6, 9, 19, 20, 21, 24, 37, 38, 39, 40, 41, 53, 54, 60, 69

ART Android Run-Time. 9

ASCII American Standard Code for Information Interchange. 40, 50, 62

DBI Dynamic Binary Instrumentation. 53, 78, 1

DBIMAFIA Dynamic Binary Instrumentation Mobile Android Format Investigation and Analysis. iv, ix, 4, 6, 7, 35, 37, 38, 40, 42, 50, 51, 53, 54, 55, 58, 60, 67, 77, 78, 79, 80, 1

DoD Department of Defense. 4, 79, 1

fbb flat buffer builder. 72, 74, 75, 77

HAL Hardware Abstraction Layer. 8

IoT Internet of Things. 1, 26, 29, 59, 79

PII personally identifiable information. 75, 1

SQL Structured Query Language. 56

US United States. 3

USAF United States Air Force. 2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2020	Master's Thesis	Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

Mobile Data Analysis using Dynamic Binary Instrumentation and Static Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

2d Lt Christopher Dukarm

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

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

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-016

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

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

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

Mobile classified data leakage poses a threat to the DoD programs and missions. Security experts must know the format of application data, in order to properly classify mobile applications. This research presents the DBIMAFIA methodology to identify stored data formats. DBIMAFIA uses DBI and static analysis to uncover the structure of mobile application data and validate the results with traditional reverse engineering methods. DBIMAFIA was applied to fifteen popular Android applications and revealed the format of stored data. Notably, user PII leakage is identified in the Hago Games application. The application's messaging service exposes the full name, birthday, and city of any user of the Hago Games application. These findings on how Hago Games uses ObjectBox library to store data in custom file formats can be applied more broadly to any mobile, IoT, or SCADA device or application using the ObjectBox library. Furthermore, the DBIMAFIA methodology can be more broadly defined to identify stored data within any Android application.

15. SUBJECT TERMS

Mobile, Reverse Engineering, IoT

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Richard Dill, AFIT/ENG
U	U	U	UU	108	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 3652; richard.dill@afit.edu