RECORD AND REPLAY OF ANDROID NATIVE CODE EXECUTION

by

URVASHI SHARMA

(Under the Direction of KYU HYUNG LEE)

ABSTRACT

Android devices are becoming increasingly popular but at the same time, constantly attracting cyber criminals. Record and replay is a popular technique for debugging and testing, and also widely used to analyze desktop malwares. Recently, several records and replay techniques have proposed to aid debugging of Android apps. However, they are not suitable to replay Android malware. Recent Android malware are increasingly utilizing native code execution that can be dynamically downloaded and run by the app to avoid detection [12]. Existing Android record and replay techniques cannot properly replay native code executions. Therefore, in this project, we aim to develop a deterministic record and replay system to analyze native code execution. Our approach can be used to understand the complete behaviors of Android applications containing native code.

INDEX WORDS: Android, JNI, NDK, VALERA, Record-Replay
RECORD AND REPLAY OF ANDROID NATIVE CODE EXECUTION

by

URVASHI SHARMA

MS Computer, The University of Georgia, 2018

BE, India, Rajiv Gandhi Proudyogiki Vishwavidyalaya, 2015

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial
Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2018
RECORD AND REPLAY OF ANDROID NATIVE CODE EXECUTION

by

URVASHI SHARMA

Major Professor: Kyu Hyung Lee
Committee: Kang Li
Lakshmis Ramaswamy
Roberto Perdisci

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
May 2018
DEDICATION

_Specially for Mom and Dad_

I want to dedicate my research work to my family and friends, and all the mentors, academic or otherwise who guided and motivate me to push the boundaries to my achievements.
ACKNOWLEDGEMENTS

I would like to thank my major advisor Dr. Kyu Hyung Lee for his guidance since the beginning. Dr. Kang Li for his constant motivation. A special thanks to Xingzi Huan (Samuel), and other team members Omid Setayeshfar, Pranav Panage for helping me out during my research. I am also grateful to Yongjian Hu, one of the author of VALERA tool for helping me understand VALERA better. And heartly thanks to my family for having faith in me.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................... v

LIST OF TABLES ...................................................................................................... viii

LIST OF FIGURES ...................................................................................................... ix

CHAPTER

1 INTRODUCTION ................................................................................................. 1

   1.1 Problem ........................................................................................................ 1

   1.2 Proposed Solution ....................................................................................... 3

2 BACKGROUNDS ............................................................................................... 5

   2.1 Recent Android Malware Behaviors ......................................................... 5

   2.2 JNI, Native Methods, Dalvik .................................................................... 6

   2.3 VALERA ........................................................................................................ 9

3 RELATED WORK ............................................................................................... 12

   3.1 Tools Approaching JNI Methods ................................................................. 12

   3.2 Android Record and Replay Tools ............................................................... 14

   3.3 Summary ........................................................................................................ 15

4 DESIGN AND IMPLEMENTATION ..................................................................... 17

   4.1 VALERA Testing on Android NDK ............................................................... 17

   4.2 Design ........................................................................................................... 17

   4.3 Implementation ............................................................................................ 21

5 EVALUATIONS ................................................................................................... 27

   5.1 Performance Efficiency .............................................................................. 28
6 LIMITATION and FUTURE WORK .......................................................30
7 CONCLUSION ...................................................................................31
REFERENCES ..........................................................................................32
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Malicious Applications with their Native Libraries</td>
<td>6</td>
</tr>
<tr>
<td>Table 2: Top 50 Applications with their Native Libraries usage</td>
<td>12</td>
</tr>
<tr>
<td>Table 3: Results of the static analysis</td>
<td>13</td>
</tr>
<tr>
<td>Table 4: Performance Evaluation for Modified and Original VALERA</td>
<td>28</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Analysis by AV-Test.org</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>C program accessing Java Methods</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Java Application accessing C++ program</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>JNI as bridge</td>
<td>8</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Overview of VALERA</td>
<td>10</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Control Flow of APIs called for JNI</td>
<td>19</td>
</tr>
<tr>
<td>Figure 7(a)</td>
<td>Calling of dvmCallJNI(Method)</td>
<td>20</td>
</tr>
<tr>
<td>Figure 7(b)</td>
<td>JNI Interception</td>
<td>20</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Modified VALERA</td>
<td>21</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Instrumenting dvmPlatformInvoke</td>
<td>23</td>
</tr>
<tr>
<td>Figure 10</td>
<td>JValue Structure for the typedefs data types</td>
<td>24</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Problem Description

Android is the most widely used mobile operating system in the world as of 2018. With the rise in smartphones, mobile development has tremendously increased since the last decade. Android provides an open source platform for development, allowing anyone to develop and deploy application on Play Store. With vast number of applications, there are some with the security breaches. According to Kaspersky Lab [7], the mobile malware evolution in 2017 was such as that they detected 5,730,916 malicious installation packages, 94,368 mobile baking Trojans and 544,107 mobile ransomware Trojans. Android security status by av-test.org [8] for 2016/2017 gives some facts and figures that state that over 4 million new malware programs have been detected in the year 2016. Their statistics table is shown as below.

Fig 1. Analysis by Av-Test.org [8]
Security of Android has always been a top concern. Most Android malware detection tools monitor and analyze Java byte code to identify malicious logics in it, however, they do not much focus on native code. However, many of the android applications contain functionality written in native code along with Java. However, there is many defensive techniques against malicious logics embedded in native code execution.

Native Development Kit (NDK) provides android application to use compile code written in other languages such as C/C++ with Android. This NDK uses Java Native Interface (JNI) which allows cross-platform functionalities. Native code can access all Android APIs that the Java code can access, as well as alter the instance of Dalvik Virtual Machine as native code can access and alter the address space of the host process. NDK and JNI are discussed in detail later in Chapter 2. NDK model allows the developer to include native code binaries as part of their application. This is usually done to include legacy libraries which are hardware dependent [9]. A recent study [4] shows that 86% of the most popular Android applications contain native code. If the attacker-controlled data are used in these interfaces without any validation, the attacker can feed payloads to abuse them [13]. This malicious code is obfuscated to bypass the anti-malware [11]. The detection of malware using static analysis usually operate on bytecode level. Therefore, the malware embedded in native code will be difficult to detect by signature-based detectors.

There still has been study about the malware infection from native code execution, but most of them focus on the detection of malware through their static techniques. Not every record and replay tool can handle native code execution. The few tools which record and replay native code are used only for debugging purpose. Therefore, there is no such tool which can analyze the native code behavior.
1.2 Proposed Solution

While there has been a lot of research on debugging android application on byte code level, less focus has been given on native code level. NativeGuard and DroidNative are some of the tools which explains the concern of threats from native code and the trend of current applications using them. While NativeGuard uses sandboxing approach and DroidNative uses Semantic based Detection method (explained in section 3). These tools emphasized on the importance of native code execution in Android. Existing record and replay tools, they aid the debugging of Android applications, but do not analyze native code behavior. The limitations of the existing software and lack of availability of such testing tool that record and replay native methods, inspired me to create a tool which can record and replay native code to analyze the program behavior. Record-and-replay is useful across the Android development lifecycle, from bug reproducing to systematic testing [1]. It is usually used for Regression testing and faster test development in software engineering. Record and Replay tools allow developers to easily record and automate the replay of complicated usage scenarios of their app [15]. This record and replay tool can also be used to analyze the behavior of bugs in a native code. The bugs can be recorded in a log file, allowing user to find the traces of malware coming from native method. The trace file will be beneficial for the developers who unknowingly use infected native libraries in their application and will probably be able to detect if the library is injected with malware or not.
CHAPTER 2

BACKGROUND

In this section, we provide background information regarding the malicious applications, base tool VALERA, functionality of the native code, how JNI is executed as well as the DALVIK environment that are used in the design and implementation of modified VALERA.

2.1 Recent Android Malware Behaviors

As discussed in the introduction section, native code in Android is considered to be as secure as java and is not considered much threat to the system. Hence, not much attention has been paid to native-code security in Android. To test if VALERA can record and replay malware behaviors, we tested some of the malwares with VALERA, some worked but most of the malware behaviors failed to be captured by VALERA. Malware App analysis was done to ensure what kind of behavior VALERA fails to record and replay. For this I tested nearly 30 APKs to verify if the malware app is recorded. For those that were recorded, I tested if they replayed. When these malicious APKs were decompiled, it generated a folder named “libraries” which contained files with “.so” extension. A library with “.so” extension indicates that a native code will execute in the application. The .so library is generated when native libraries are used with JNI (Java Native Interface). Table 1 shows the data when some of the known malware applications were decompiled using apktool and certain .so libraries were found.
Discovering the patterns of the .so libraries found in such malware applications and VALERA failing on recording and replaying, we decided to test the non-malware applications using native code. First, I tested real-world application System Lite which confined native libraries on VALERA. It was expected that VALERA will not successfully record and replay this application as VALERA does not account for native methods. Therefore, to understand the working of AOSP and native methods, I first dig in details about JNI, native code and Dalvik environment.

2.2 JNI, Native Methods, Dalvik

To understand the better working of VALERA on .so library, I wrote my own application with native code using JNI. This section explains JNI, native method and Dalvik in detail.

JNI is the native programming interface that is a part of JDK. When JNI is written in programs, you can ensure that your code is completely portable across all platforms. This invocation API allows you to embed JVM (Java Virtual Machine) into native apps.

Native Methods are those which use JNI calls in the java program. These native methods can create, update and access Java objects [2]. Android uses Native Development Kit for developing applications with JNI. Figure 2 shows how C program links with Java libraries,
call Java methods and use them. Whereas Figure 3 shows how Java application calls native language function.

Hence, it is said that JNI acts as a bridge between Java application and native program written in different language than Java to allow cross platform. Figure 4 shows how native app on Android works if the native library is written in C language.
Figure 4 explains the JNI bridge across Java application and C Application. JNI defines a way for the bytecode that Android compiles from managed code to interact with native code [6]. Native libraries in Android are loaded using System.loadLibrary() method with argument as .so file of equivalent CPP file.

VALERA works on Android 4.3.1 which uses Dalvik. Dalvik is a virtual machine (VM) Google’s Android operating system that allows execution of android applications. Programs written for android applications are compiled to bytecode, which is later translated to Dalvik bytecode and then stored in .dex(Dalvik Executable) files. Dalvik is now discontinued and was used until Android 4.4 “KitKat”. The advance version of Dalvik, ART (Android Run Time) is continued in current Android operating system versions.

Dalvik VM requires fewer, but more complicated Virtual Machine instructions compared to Java VMs. Dalvik programs are written in Java using the Android Application programming interface (API), which is compiled to Java bytecode, and converted to Dalvik instructions as necessary.

2.3 VALERA

VALERA stands for Versatile yet Lightweight record and replay for Android. As the name says, VALERA is an approach and tool for versatile, low overhead, record and replay for Android apps. As mentioned in [1], VALERA records and replays smartphones apps, by intercepting and recording input streams and events them with minimum overhead, replaying them with almost exact time as the base time of an application. Unlike other software tools, VALERA keeps performance overhead low, on average of 1.01% for record and 1.02% for replay [1]. The tool follows the stream-oriented approach for
recording and replaying Android apps. For record and replay system for Android features VALERA is designed to meet these desiderata [1]:

1. It supports I/O (sensors, network) to record information with high accuracy and replay popular full featured apps.
2. It accepts APKs as input- as accessing APK is easier rather than getting access to the app source.
3. Debugging with VALERA can be done working with phone directly, rather than working on Android emulator which only supports limited subset of sensors while all the sensor inputs are available on phone device.
4. It has low overhead to avoid disturbing the app’s execution.
5. It requires no hardware, kernel, or VM changes.

Figure 5. Overview of VALERA [1]
VALERA consists of runtime component and an API interception component. As in the above figure, the instrumented app runs on top of the Android framework, which further runs on unmodified Dalvik VM and the kernel. It performs App Instrumentation by rewriting bytecode, which is used to intercept communication between the app and Android Framework to record in a log files. This Android Framework instrumentation is used to log and replay the event schedule. Valera runtime consists of record and replay code and the log files. This code does not require any extra threads or processes. Also, to achieve to clarity, the tool omits the VM copies of the app, which are running in their own address space.

The second component of API interception is generated automatically through app rewriting, based on interceptor specification.

*Limitation of VALERA:

Now with the current approach of VALERA, it does not meet few desiderata. The tool achieves replay precision by eliminating sensor input nondeterminism, network nondeterminism, and event schedule nondeterminism [1]. Nevertheless, VALERA does not record and replay all system state. Since VALERA does not deal with the Kernel or the VM modification, it does not record memory access or the VM instruction. Based on this VALERA also does not support record and replay of JNI calls.
CHAPTER 3

RELATED WORK

3.1 Tools approaching JNI methods

To study more about how third-party Native libraries, I came across certain tools focusing on the approaches to protect Android applications from Native libraries.

Native Guard:

Sun and Tan use their tool NativeGuard to protect Android application from Third-Party Native Libraries. The paper [4] believes that the current native components are not well managed by Android’s security. Therefore, NativeGuard is a security framework which isolates the native code to a standalone application from other components of an application. NativeGuard thereby eliminates the unnecessary privileges given to this standalone application. The developers of NativeGuard claim that if the native code is exploited it may potentially access/modify any data within the process boundary, leading to possible privacy violation or application malfunction. According to their survey, usage of third party native libraries in applications can be categorized as in Table 2.

<table>
<thead>
<tr>
<th>Category</th>
<th>Apps</th>
<th>Apps with native libs</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social</td>
<td>7</td>
<td>7</td>
<td>100%</td>
</tr>
<tr>
<td>Communication</td>
<td>5</td>
<td>4</td>
<td>80%</td>
</tr>
<tr>
<td>Gaming</td>
<td>19</td>
<td>18</td>
<td>95%</td>
</tr>
<tr>
<td>Entertainment</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Other</td>
<td>17</td>
<td>12</td>
<td>71%</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>43</td>
<td>86%</td>
</tr>
</tbody>
</table>

Table 2: Top 50 Applications with their Native Libraries usage [4].
With their analysis, they found a total of 200 native libraries, showing average of 4 native libraries used per app. Native Guard utilizes the natural ability of underlying Linux Kernel in Android to isolate untrusted native code. It uses sandboxing to relocate Native Lib to second sandbox besides the original. This second sandbox runs in a separate process. Therefore, longer direct access is not needed as before. NativeGuard follows an approach to isolate native code in a second application providing less permissions. After the isolation the two applications interact via interfaces defined by Android Interface Definition Language. NativeGuard later adds mandatory smali code to the launcher class(es) for service binding. In the end two applications are built as the output. Hence, NativeGuard is a security framework which isolates native code libraries into a non-privileged application [4]. Native Guard is one of the earliest work done on threats via JNI, which provides a good motivation to focus more on security research in native code in Android.

*Going Native:*

Authors of the paper Going Native [14] collected 1,208,476 distinct—different package names and APK hashes which they continuously downloaded from 2012 to 2014 and found the result in Table 3.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>267,158</td>
<td>Native Method</td>
</tr>
<tr>
<td>42,086</td>
<td>Native Activity</td>
</tr>
<tr>
<td>288,493</td>
<td>Exec Methods</td>
</tr>
<tr>
<td>242,380</td>
<td>Load Methods</td>
</tr>
<tr>
<td>22,515</td>
<td>ELF Files</td>
</tr>
<tr>
<td>446,562</td>
<td>At least one of the above</td>
</tr>
</tbody>
</table>

*Table 3: Results of the static analysis [14]*
Going Native follows NativeGuard approach to dynamically analyze android applications to study the usage of native code and automatically generate native code sandboxing policy.

*DroidNative:*

DroidNative follows approach based on semantic detection of Android Native Code Malware. They view native code as machine code which is directly executed by a processor, opposite to bytecode which is executed in a virtual machine such as JVM or Dalvik VM [9].

It performs its analysis on intermediate representation of an Android Application’s binary code. To achieve this, they use Malware Analysis Intermediate Language (MAIL). Thereby, it uses control flow with patterns to reduce the effects of obfuscation; using the language MAIL to provide automation and platform independence; allowing real time malware detection [10].

### 3.2 Android Record and Replay Tools

These following are tools that perform the Record and Replay testing of Android applications.

*Jalangi:*

Jalangi is a framework for writing heavy-weight dynamic analyses for JavaScript [18]. The tool performs dynamic analysis of a program in two modes: online and offline. It performs the selective record and replay technique, meaning, the tool selects the subset of JavaScript source to instrument it for record and replay. During recording, the entire application is executed with its instrumented and non-instrumented code, but Jalangi only replays the instrumented code. During this replay phase, Jalangi performs shadow
execution of instrumented code. Hence the tool can only perform dynamic analysis of the instrumented code and cannot analyze the behavior of uninstrumented or native code.

Mosaic:

Mosaic is a cross platform tool, time accurate record and replay tool for Android based mobile devices [20]. Mosaic however ignores the input devices which do not correspond to the touchscreen. Throughout the recording it sends the sequential events of touchscreen input events to a file. These input events are further used to convert into primitive interactions (i.e. press, move or release). These interactions are stored as a platform agnostic representation so that Mosaic can interpret each touch input event. The tool has its limitations in these scenarios: 1) If Applications UIs is either time-variant or random which may require different input each time with every user. 2) If user uses customized keyboard on the application. 3). To provide accurate record and replay, the tool fails to provide the accuracy on older smart phones as platform’s performance is the requirement.

MobiPlay:

MobiPlay is a client-server system allowing record and replay of mobile applications. It runs the target application on the server, while displaying the real-time GUI on the mobile phone. Mobiplay is the first tool providing record and replay on Application level, but the server works on x86-based environment. Mobiplay provides time overhead of roughly 2% - 4% due to following factors: 1). After injecting one event, MobiPlay waits for a period time interval between two events to inject second event using thread.sleep() method 2). It takes time to set the time information of an event based on current time. 3). Reading the data from the disk requires time as well.
3.3 Summary

The tools helped me get a better understanding of how JNI methods can be threats and can be prevented, as well as the importance of record and replay in software testing. While the first category provides different approaches to restricts the permission given to the native methods by isolating native code, second one provides different approaches to record and replay on various levels and type of resources each tool requires. But these tools limit their record and replay method approaches to non-native methods. While Jalangi and Mosaic does not replay native methods, Mobiplay does; however, the remote execution of server makes it not suitable to analyze native code behavior. Henceforth it does not satisfy our concern. Subsequently learning various tools, I chose to use VALERA as my base tool to extend its approach to record and replay native code executions for following reasons:

a) It follows stream-driven approach which provides accuracy.

b) The tool performs its API instrumentation on framework level.

c) It records and replay a wide range of popular APIs

d) VALERA does not require source code of an application.

e) It provides low performance overhead with high accuracy.

f) VALERA is able to run on emulator as well as mobile devices.
4.1 Valera Testing on Android NDK

So far, we know Valera eliminates sensor input non-determinism, network nondeterminism and fails to record and replay system calls. Here we focus on how VALERA is unable to record and replay JNI methods. After malware analysis, native code is the threat to Android system which needs to be covered. I tested couple of applications such as System Monitor and Monitor Lite. Both applications used .so libraries. Since, it was known that VALERA cannot record and replay the execution of native methods, I designed my own Android NDK which uses JNI. It is simple application using rand () function in the cpp file. The code is as follows.

After writing this application, when we execute our app on emulator using Valera record command and replayed it, it failed. While debugging the Valera Log and the logcat generated by “adb logcat”, it was concluded that VALERA does not encounter any calls involving native code.

4.2 Design

VALERA records the event schedule by logging each message sent and each message processing operation into a trace file. Every time a thread sends a message, VALERA record this operation as a < etype, eid, pid, tid, type, looper, caller> tuple [1].
The etype indicates the type of internal or external event, eid stands for the unique identifier (TAG) given to each kind of method, pid and tid are process ID and thread ID respectively, type shows whether this event is a handler or a runnable action, looper is the target Looper object, and caller records the caller method that has created this event message. During replay, an event is matched with the recorded schedule if the tuple matches. If the tuple matches, then the procedure returns true indicating that this event can execute.

Using VALERA concept of record and replay, I chose to modify Dalvik to record and replay native calls too. As we know the second component of VALERA is API interception, hence here I am instrumenting APIs used in execution of JNI method calling. VALERA instruments each API manually and sends the data to Log File which is further used to replay the events. When run script is executed it sends the information to ValeraConfig.java file, where it sets the package name of the application, specifies the thread used, set the enumeration of which mode is used i.e. replay or record or none and so on. The main wide-ranging API which VALERA targets are Network API, Touch Event API, Geolocation, Camera, View, Audio Manager. It also instruments the Looper.java and Handler.java to receive the messages send by the application.

To follow VALERA’s approach to extend the tool, it was necessary to have a better understanding of the control flow diagram of JNI working on VM level. Figure 6 shows control flow to give better understanding of JNI working in AOSP.
When an NDK android application is executed it calls the API in System.java to load library (.so). The first API in the control flow loads and links the library with the specified name. API `dvmLoadNativeCode` load native code from the specified absolute pathname. This library is associated with the specified class loader. API `dvmResolveNativeMethod` resolves a native method to register this method. The API acts as a prototype for DalvikBridgeFunc and calls the API `dvmUseJNIbridge`. The latter API overload instants of a method to point at the actual function. Figure 7 (a) shows before creating a bridge `dvmUseJNIbridge` checks if the method is native or not. If it is, it calls `dvmCallJNIMethod`. 
In Figure 7 (b) dvmPlatformInvoke takes the arguments and execute the native binary code which has been loaded already. The function takes one of the argument as pResult (pointer to the JValue structure pointed to the data type of the return value of the native method) which is null initially. After the execution of dvmPlatformInvoke, pResult points to the return value of the native method. Depending on the return type of native method, pResult stores the result of native code execution.

After learning the basic control flow of native code execution on VM level, I intend to instrument dvmPlatformInvoke. VALERA creates it's two log files, one for inputEventManager and other for IOManager. Since there is no logging for VM level calls, I created my own log file with tuple same as VALERA such as <Native, eid, methodName, tid, pResult>. The tuple is logged in a file during record mode and while replay it matches the tuple schedule to generate the pResult value from the log file. To be able to capture processed native code, I instrumented return values of dvmPlatformInvoke for successful record and replay of my tool.

The modified VALERA design is overviewed is shown in Figure 8.
4.3 Implementation

As discussed in above section on how to instrument native method API, I chose to modify few files of VALERA’s version of AOSP, such as valera.cpp, valera.h, Jni.cpp, Globals.h, java_lang_reflect_Method.cpp. To generate my log file for JNI recording, I modified the VALERA script for capturing these natives. The implementation strategy of each file is listed below:

**Globals.h:**

As the name suggests, I created some global variables such as `valeraJniFile` and file object `valeraJniLog` in order to continuously log schedule events and use the same file to read the events during replay.

**java_lang_reflect_Method.cpp:**

In method name `Dalvik_java_lang_reflect_Method_invokeNative`, VALERA reads the configuration of the application tested. Based on the module, I configure the path of my log file as:
gDvm.valeraJniFile=std::string("/data/data/") + gDvm.valeraPkgName + "/valera";

If VALERA is enabled, the log file is set to gDvm.valeraJniFile+="/jni.txt"

**Jni.cpp:**

This is the file where I handle the API which invokes native code. According to AOSP documentation [21] _dvmCallJNIMethod_, forms a bridge between interpreted code and JNI native functions. The method basically converts an array of primitives and references to C-style functions arguments [21]. This bridge takes four arguments: the array of parameters (const u4* args), a place to store the function result (JValue * pResult), the method to call (Method* method) and a pointer to the current thread. To enable record and replaying, instrumentation is done in such way:

```c++
1.  int mode=gDvm.valeraMode;
2.  switch(mode){
3.    case VALERA_MODE_NONE:
4.      dvmPlatformInvoke(env, (ClassObject*) staticMethodClass,
                        method->jniArgInfo, method->insSize, modArgs, method->shorty,
                        (void*) method->insns, pResult);
5.      break;
```
Figure 9: Instrumenting dvmPlatformInvoke

```c
  case VALERA_MODE_RECORD:
    1. dvmPlatformInvoke(env, (ClassObject*) staticMethodClass,
                     method->jniArgInfo, method->insSize, modArgs, method->shorty,
                     (void*) method->insns, pResult);
    2. if(strcmp(method->name, "gDvm.MethodName") == 0)){
    3. recordJniCall(pid, self, method, pResult);
    4. break;
  case VALERA_MODE_REPLAY:
    1. if(strcmp(method->name, gDvm.MethodName) == 0))
    2. replayJniCall(self,method,pResult)
    3. break;
```

**Record**

To record the pResult of test application, I intended for method->clazz->descriptor to be equal as the method name defined by the VALERA. Hence using gDvm.methodName, I match the method->clazz->descriptor. If the condition is true and VALERA is enabled with record mode, I am able to record the pResult generated with the dvmPlatformInvoke API.
Replay -

When Valera executes in replay mode, \texttt{gDvm.valeraMode} is set to \texttt{VALERA\_MODE\_REPLAY}. As we know every time native code is executed, \texttt{dvmCallJNIMethod (const u4* args, JValue* pResult, const Method* method, Thread* self)} is called. Here instead of calling \texttt{dvmPlatformInvoke}, we read the log from our log file and store the read value to \texttt{pResult}, instead of calling machine code again.

The bridge functions aren't called directly from elsewhere in the VM. A pointer in the Method struct points to one of these, and when a native method is invoked the interpreter jumps to it [23].

This returned value is stored as \texttt{pResult}. \texttt{dvmPlatformInvoke}, invoking the method arguments, and performing the execution of the method body defined in c++. It converts \texttt{argc argv} into a function call, making it platform specific. \texttt{pResult} is a pointer to \texttt{JValue} structure as shown in Figure 13:

\begin{verbatim}
typedef union jvalue {
    jboolean z;         jbyte  b;
    jchar   c;          jshort  s;
    jint    i;          jlong   j;
    jfloat  f;          jdouble d;
    jobject l;
} jvalue;
\end{verbatim}

Figure 10: JValue Structure for the typedefs data types
valera.h:
Define the tag as unique Id for the particular API. For my tool I have defined eid for natives as

```c
#define VALERA_TAG_MYJNI 93
```

Valera.cpp
In this file, I have implemented recordJni and replayJni methods. During record mode, it creates the tuple as described in the design mode and sends to the ValeraJniFile. The events are orderly sent it to the file. During replay, it matches the tuple, with eid, pid and tid and accordingly fetch the result and storing it to pResult, instead of dvmPlatformInvoke API.

Run.sh
At client side of VALERA, after VALERA is done recording, it fetches the Log File created on the relative path of application directory, to the local machine. During the replay push this recorded file to application directory, allowing it to read the events and results from the log.

JNI calls are different than the normal android application APIs. When the android device starts, it loads many .so files at the system level which thereby calls Jni.cpp and each time dvmCallJNImethod is called for the background processes. But when VALERA records and replays event sensor API, or network API, motion API etc then these APIs becomes application specific and are only executed when user run the application which needs to be record and replayed. For example, if I have written an application with random number generator in Java, VALERA instruments View.java and Random.java. So, whenever a user clicks a button, the event of touch is recorded, and value generated by Random method is further recorded. As VALERA does not mess with Kernel level methods or vm level
methods, it gets easier to record such API. Now here is the challenge to instrument Jni.cpp as the classes at Kernel level calls this file too. Hence it becomes necessary to specify your native library which you want to record.
CHAPTER 5
EVALUATION

I have carried out my evaluation in two stages to test modified-VALERA’s functionality and performance. First, I developed 4 applications using Native Development Kit. To test the efficiency of my tool, I tested it on my applications and few real-world applications. Second, for performance testing I have executed the applications with original VALERA and the modified version of it. Experiments were conducted on a Samsung Galaxy Nexus 4.3.0, Linux Kernel version 3.15.1 on a quad core ARM CPU@4GHz.

Functionality Testing

I have manually created test native applications, which demonstrates calling of a native method with return type of primitive data type. The application involves DiceRoll, TossTheCoin, Random Number Generator. The applications are used to test the efficiency of record and replay of native code execution, where the output is uncertain.

Real-world Application Testing

I collected few real time applications which execute native code to test on my tool. Since VALERA is bound to certain APIs and .so files do not reflect much on user interface, it was difficult to collect many APIs required for this efficiency testing. My tool only works for few real world native applications as I do not account for complex data structures used in native method calling, which I intent to do in my future work.
5.1 Performance Efficiency

To demonstrate the importance of record and replay of native data, I ran modified VALERA on few applications. Applications are chosen such a way that they satisfy most of the primitive data types the tool focuses on. Table 4 lists the apps I chose for my evaluation.

<table>
<thead>
<tr>
<th>App</th>
<th>Modified VALERA</th>
<th>VALERA</th>
<th>SPACE</th>
<th>VALERA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Record</td>
<td>Overhead</td>
<td>Replay</td>
<td>Overhead</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>Time (sec)</td>
<td>Overhead</td>
<td>Time (sec)</td>
<td>Overhead</td>
</tr>
<tr>
<td>Random Number Generator</td>
<td>10.24</td>
<td>0.015</td>
<td>10.56</td>
<td>0.047</td>
</tr>
<tr>
<td>Dice Roll</td>
<td>12.45</td>
<td>0.075</td>
<td>12.75</td>
<td>0.108</td>
</tr>
<tr>
<td>Toss the coin</td>
<td>13.53</td>
<td>0.127</td>
<td>14.35</td>
<td>0.191</td>
</tr>
<tr>
<td>System Lite</td>
<td>17.67</td>
<td>0.077</td>
<td>18.4</td>
<td>0.12</td>
</tr>
<tr>
<td>CharThe Mood</td>
<td>10</td>
<td>0.098</td>
<td>11.23</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 4: Performance Evaluation with modified VALERA
In summary, the experiment demonstrates that the approach to record and replay the native methods succeeds for these applications. However, with certain limitation, it only works for the predefined JValue Union structure. As seen in Table 4. Log Size only varies a little. However, with overhead, recording overhead does increased compared to VALERA, but for JNI methods, overhead of VALERA is far much worse than my tool.
CHAPTER 6
LIMITATION AND FUTURE WORK

The extended tool only works for inbuilt methods of JNI and primitive data types. These includes jint, jchar, jlong jfloat, jbool, jshort and so on. It registers all the native method coded in the application. But the application does not record much complex data type such as arrays, class, or user defined structure. The instrumentation performed currently only works for the JValue union structure which is predefined by the Android Open Source Project.

Another drawback of the tool is that it only works on Dalvik environment. In my future work I plan to extend the tool on ART environment as well.

In future work, I intend to extend the tool for these complicated data structures created by the user. The thesis talks about the threat from the native code, however as of now it is not tested on Malware Android Applications. Yet it provides the technique for future to be able to work on applications available on Play store or on third party. As researched currently, Eagle Eye is the new tool, an Xposed and adbi based model which overcomes the limitation of my tool by hooking every method call. All the informed hooked methods will be logged as the output.
CHAPTER 7
CONCLUSION

Record and replay is a useful technique to analyze software behaviors including malicious ones. Several record and replay techniques have been proposed. However, they cannot deterministically replay for native code execution. For example, VALERA is a lightweight tool for record and replay of Android apps, but it does not record JNI method properly. Thus, cannot be used to analyze Android malware that contain native code. In this work, I extend VALERA to make it support to record and replay JNI methods. Although the tool does not modify the system level or VM level API, I intended to modify the VM level API to instrument JNI calls, and thus successfully record and replayed the native code of an Android Application. The evaluation results show that additional time and space overhead to record JNI methods are negligible.
REFERENCES


[12] JNI.h specification
https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html


[22] Eagle Eye: https://github.com/MindMac/AndroidEagleEye

[23] AOSP https://android.googlesource.com/?format=HTML