7-1-2016

Automated code extraction from packed android applications.

Abhishek Vasisht Bhaskar
Syracuse University

Follow this and additional works at: http://surface.syr.edu/etd
Part of the Engineering Commons

Recommended Citation

This Thesis is brought to you for free and open access by the SURFACE at SURFACE. It has been accepted for inclusion in Dissertations - ALL by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.
Abstract

Software packing is a method employed by malicious applications to hide their original intent. Extracting the original intent of an application from its application bundle, whether to perform a security analysis on it, to search for security flaws (or bugs) or simply for educational purposes is a key requirement for the security community. With the fluidity provided by the Android app store coupled with a complete application-framework based environment for a malicious user to employ as an attack space, it is of great importance to examine Android applications and extract their intent. For basic applications, simple reverse engineering tools can be used to extract a semantic view of the application very close to the original source code of the application. However for applications, which have been deliberately packaged/packed in such a way that their original intent cannot be extracted by simply reverse-engineering them, we need a more intricate procedure to extract enough information to be able to reproduce the original intent of the application. These applications are packaged such that the actual code is hidden/encrypted and only during run-time is the actual code unpacked and executed. To unpack such applications, we present DroidUnpack, a tool based on dynamic program analysis, which is able to extract the original intent of the application, generically. DroidUnpack is designed by exploiting some fundamental features of the Android Runtime which cannot be mutated by a malicious user to unpack the application. We also attempt to alleviate tedious manual analysis required by a user to analyze different types of packed applications, by providing a generalized tool which is able to unpack android applications, regardless of the packing technique used.
Automated code extraction from packed android applications.

by

Abhishek Vasisht Bhaskar

B.E., PESIT, Bangalore, India, 2014

Dissertation Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering.

Syracuse University

July 2016
Acknowledgements

Most importantly, I would like to thank my advisor, Dr. Heng Yin, who has guided me at each step of the problem and provided constant motivation and push to keep going. Thank you professor, for helping me grow and encouraging my work at all times. Secondly, I would like to thank my mum and dad, although not in the same content they have been giving be constant moral support for all the time during my thesis research. Thirdly, I would like to thank Dr. James Fawcett, Dr. Kevin Du and Dr. Richard Tang who took time out of their schedules to be a part of my defence and provided me with a lot of insight and advice on the problem. Lastly, I would like to thank all the members of the System Security Lab headed by my advisor Dr. Heng Yin, all of you have been very helpful and forthcoming whenever I had any technical doubts.
Contents

1 Introduction 1

2 Background and solution overview 3
   2.1 Android application package (APK) packing 3
   2.2 APK unpackers 4
   2.3 Solution Overview 7
   2.4 Contributions 8

3 The Android run-time (ART) 9
   3.1 History 9
   3.2 What is the new Android run-time (ART)? 11
   3.3 \texttt{libart.so}: the heart of ART and its execution mechanism 16
   3.4 Final words 17

4 DroidScope\{ART\} 19
   4.1 Overview and older Droidscope 19
   4.2 The new Droidscope with ART support 20
      4.2.1 Recovering Native semantics 20
      4.2.2 Recovering Java semantics: Introduction 22
      4.2.3 Compiled ART methods 23
      4.2.4 Interpreted Java methods 26
List of Figures


3.2 OAT File Layout ................................................. 14

3.3 Framework OAT file memory dump .............................. 15

3.4 Quick invoke, stack and register layout .......................... 18

3.5 32-bit code layout of the Dex::CodeItem data structure which holds the dalvik byte-code. ........................................ 18

6.1 Behavior of the Alibaba packer. .................................... 38

6.2 Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin. .................. 40

6.3 Behavior of the Baidu packer. .................................... 41

6.4 Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin. .................. 42

6.5 Behavior of the Bangcle packer. .................................... 43

6.6 Comparing the JAVA API traces between the unpacked and the packed application output both obtained from our unpacker plugin. .................. 45

6.7 Behavior of the Ijiami packer. .................................... 46

6.8 Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin. .................. 47

6.9 Behavior of the Qihoo360 packer. .................................... 48
List of Tables

6.1 Overall Comparison .................................................. 37
Listings

4.1  Sample code showing how guest data structures are read from memory . . .  24
5.1  DexFile::CodeItem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
A.1  Source code from the DroidUnpack plug-in. . . . . . . . . . . . . . . . . . . 54
Acronyms

APK Android application package
ART Android run-time
VMII Virtual machine introspection
SLOC Source lines of code
DVM Dalvik virtual machine
AOT Ahead-of-Time
OS Operating System
GC Garbage Collector
ASM Assembly
API Application Program Interface
TB Translated Block
CB callback
BB basic-block
Chapter 1

Introduction

Software obfuscation or Run-time packing is a intricate tool used by attackers and software vendors alike to protect their code. Although it can be an absolute boon for software vendors, helping them protect their closed source code base, it can be an absolute nightmare for security analysts when they encounter malicious application which are obfuscated or packed. Simple reverse engineering of such applications proves ineffective and more complex mechanisms are needed to extract meaningful code belonging to these applications.

Binary packing on desktop computers, being a very old problem has been extensively studied since its discovery and various solutions have been designed to accordingly handle these packed application and extract meaningful source code from them. Although this is the case, the problem of handling binary samples from the wild was scarcely addressed as highlighted very recently by "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers"[17].

With the onset of smart-phones, there had to be a reinvention of the run-time environment to adapt to a completely different user interface and hardware structure than a desktop computer. On Android, a virtual-machine based sandbox-styled interpreted environment based on dalvik byte-code, very similar to java byte-code was designed. As these systems were designed, new packing techniques were introduced and classic binary
unpacking solutions were no longer applicable to them. Various projects in the recent years have attempted to unpack these android applications to diverse degrees. Although a portion of them produce very accurate results, their extraction processes are based on explicit packing features, modeled around some of the state of the art packers available for Android applications. Dynamic analysis based unpackers usually insert hook-points in the run-time and/or kernel source code to extract files from memory when certain trigger features are met. Albeit they result in successful code extraction for applications packed with any of the known packers, a smart malicious agent could easily subvert these detectors by changing the packing design ever so slightly. Moreover, with the advent of the ART where most/all of the dalvik functions are translated into native code, the problem of unpacking becomes even more complex as simple tap points fail to provide complete coverage of the executed code. We take a brand new perspective to solve the problem of generic code unpacking by considering factors which cannot be manipulated by these packers. We register for key events which represent java/dalvik method dispatch points in perspective of the run-time and excerpt information for the particular method from the guest memory by, reading ‘state information from run-time data structures for dalvik interpreted methods’ and ‘native code from the oat file for ART native methods’ both of which remain accurate regardless of the packing technique used. We first design and implement a dynamic analysis platform for the new ART based on "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis" [18], which provides various Virtual machine introspection (VMI) tools and finally we present DroidUnpack, a plug-in, about 703 Source lines of code (SLOC) of C++, which performs generic code extraction from packed android applications.
Chapter 2

Background and solution overview

2.1 APK packing

Run-time packing or Executable compression is a process where the the code and/or data of an application is compressed/encrypted to various degrees and a run-time element, usually a shared library or such is used to dynamically decompress/decrypt the original code and execute it. This process is employed by malicious users to hide their program’s original intent. Even after years of arduous research on trying to propose a generic method of unpacking, Ugarte-Pedrero et al. (17), after conducting a thorough study on the complexity of run-time packers showed that a great majority of samples employ a multi-layered packing mechanism, whereas most solutions only expect simple single-layered code unpacking and are ill-equipped to handle different/complex packer designs. This paper importantly highlights the lack of a stable generic unpacking scheme.

While still based on the same core principles as its binary predecessors, android APK packers have starkly different designs. Most of the run-time packers, start with the APK which is merely a set of .dex files and resources corresponding to the particular application and encrypt each of the .dex file and create a new APK which contains a single .dex file (which would act as a launchpad for the application), an obfuscated native library and file-
chunks corresponding to the original .dex files. The new application starts up from the single .dex file which then loads the obfuscated native library. This library is the main unpacking agent which performs all the necessary steps to correct the file-chunks to form a verifiable .dex file/s representing the original application, load it into memory and start executing the application. While this a commonly followed design, most packers differ in that they 1. employ different ways to obfuscate/unobfuscate .dex files, 2. different ways of launching the application which affects the complete execution pattern of the application... .To cover their tracks these packers employ various techniques like 1. deleting any corrected .dex file they drop into memory as soon as it is loading 2. skewing the .dex file backing data structure of the run-time in memory to hinder debuggers 3. hooking various system functions to detect if being tracked... which make it especially hard for a security analyst to deduce their original intent. Many of these packers now fully support the new ART which is much more sophisticated than the older dalvik run-time causing a bigger challenge. In essence, apart from the fundamental principle of the ART mechanisms which are built in, anything that the packer can control can be fair gain for implementing packing features. The next section talks about some of the unpackers which exist and goes on to highlight the need for a generic unpacking mechanism.

2.2 APK unpackers

There have been many projects and publications alike attempting to solve the problem of packed android APK and many of them are successful for particular samples sets. In general, for a packed APK they start off by manually investigating the behavior of the packed application under execution, noting down techniques used by the packer. Once this is done, some of them propose an automatic framework to extract .dex files and others manually do so, both relying on the behavioral aspects of the packer which they deduced in the previous step. This investigation is then repeated for a suit of know packers. Since these
packed applications are virtually impossible to extract using any known static analysis based approaches, all the unpackers (which intend to handle complex packed applications) are based on different dynamic program analysis techniques. Nasim et al. (14) perform unpacking by performing a memory dump when a new module is loaded by the application, using either a kernel module or a ptrace based method and then a python script to parse the memory to find the .dex file corresponding to the application and extract it. Modern packers are very advanced in that they have anti-debug features built into them to detect ptrace based tracking methods. They also hook common functions used to read into the memory and suspend the process if they observe that they are being tracked or another program is attempting to read their memory space, hence easily evading this unpacking scheme. Kim et al. (13) develop another such a similar unpacker project which attempts to dump the memory but instead uses a method whereby they change the source code of the DVM and add hook functions into a function dvmFexFileOpenFromFd which is used to load the .dex file, and at that point dump the memory belonging to the .dex file data structure passed onto the function. .dex files are often mangled by a packer and during the time of loading to memory are not completely reliable, in that their contents are not accurate and cannot be assumed to be complete. The DVM does not perform any code verification, but instead just checks if the different headers and offsets in the .dex file hold good when loading it. Some packers take advantage of this and have a child process dedicated to correcting the .dex file during the runtime. This would mean that the a .dex file collected during load time may be incorrect. "Android packers: facing the challenges (19) is another work which uses "LiME" (16) to read into the memory and "volatility" (11) plug-ins to perform memory forensics on the collected memory dumps. "General unpacking method for Android Packer(NO ROOT)" (15) is another project which hooks functions, in their case they hook different functions for different packers, to perform a memory dump. Keeping all these problems in mind, Zhang et al. (20) take a slightly different approach to unpacking by identifying known packers using 1. inserted classes 2. location_ for ART and fileName
for DVM. The former is used to identify the packer while the latter is used to get an idea of the location of the .dex file for the dalvik run-time or the ART to extract it.

These solutions suffer from the same issues where the packer’s features need to be studied before hand to get a fair idea of what functions are to be hooked etc. They also suffer from the fact that more advance packers perform complex selective code unpacking, which means that these unpackers unload a .dex file into memory and start up the application, but the dex code of all of the methods in the .dex file are encrypted or obfuscated, and a runtime library belonging to the library performs selective unpacking where in it decrypts the .dex code of the method right before this method is called defeating any unpacking attempts where the unpacking scheme which performs a memory dump at only one specific point in the application life-time.

With all these things in mind, there was a recent work by Bodong et al. (10) which attempts to address the problem of generality in unpacking. They go about their work by hooking into all the functions which are responsible to interpret dalvik code in the DVM (source code) and in their callback function, they read the DexMethod data structure and extract dex code specific to each method. Although feasible, this scheme will not work for the ART because of several reasons,

1. There is very minimal interpreted code in ART and a majority of the code is compiled Ahead-of-Time (AOT), hence hooking any particular method in the ART library will not provide complete code coverage.

2. Once an ART native method is dispatched from the run-time from a particular function, calls to other ART native methods, within that module, from that point on, need not trap back to the run-time
2.3 Solution Overview.

There are many challenges facing APK unpacking as we saw in the previous sections. Packed samples are completely immune to static analysis. They have anti-debug features which render unpackers which attempt to read .dex files from the memory via system calls or ptrace based methods useless. Some packers mutate the .dex file contents in memory, which means that even after getting a memory dump, an analyst cannot successfully find or extract the actual code. Finally, with 69% of the android users now using KitKat Operating system or above, all of which are based on the new Android run-time (ART), an unpacker must be able to support it.

To accommodate to these shortcomings we look at the problem from a different perspective. Regardless of what packer is used to pack an application, what features it implements, because applications on the android phone are run and managed by a standard ART run-time, they have no control of the execution engine of the run-time. For ART compiled native methods, there are some data-structures which hold important data, like the offset of these methods in a module and for interpreted .dex methods, there are data-structures which hold the dalvik byte-code corresponding to each method, in the run-time, both of which are not in the packer’s control. Execution of an android application (albeit benign or malicious), whose core logic has been written in Java and compiled into an APK and then packed, has to resemble that of the unpacked application and to satisfy this, the packer cannot mutate certain structures in the ART run-time at any cost.

Acknowledging these facts we first design a system which is able to provide us with a platform to perform unpacking. We build this platform as an upgrade to [18], which is based on the Android emulator (which is based on QEMU). It is designed such that it can support the new Android emulator as well as the recent ART run-time. This emulator based dynamic analysis tool provides us full control of the guest system and all the necessary features to perform the unpacking. The unpacking process itself is done by carefully studying the ART run-time and extracting information from some key data-structures in the run-time at key
events (/hook-points) during the execution of the application. We are able to produce a code
extraction of all the native/interpreted methods that belong to the application during the
lifetime of the application accurately defeating run-time packers, hence providing a security
analyst with a block by block trace of the application for further analysis.

2.4 Contributions

1. A complete dynamic analysis platform which supports various virtual machine intro-
spection features like

   (a) native-call tracing
   (b) native-instruction tracing
   (c) java function tracing
   (d) memory read/write tracing ...  

for the new ART run-time, based on Droidscope, a similar such tool for the DVM run-
time .And in doing so a good overview of the working of the Android run-time (ART)
for interested researchers.

2. An unpacker, implemented as a plug-in for the above-mentioned tool, which performs
   a very generic unpacking procedure which is able to successfully extract accurate code
   from any packed application, regardless of what packing mechanism was used.

3. A case study of some special known packers

4. A platform for an security analyst to perform further analysis (apart from just code
   extraction) on the behaviour of these packed applications from the wild.
Chapter 3

The Android run-time (ART)

Before we begin speaking about the unpacker, it is important to understand the new Android run-time (ART) to help the reader in getting a firm grip of this system as well as lay the ground for the later sections. The Android run-time (ART) is the application run-time environment used by the Android operating system since version 4.4 "KitKat".

3.1 History

Before ART the android Operating System (OS) was based on a process virtual machine, whereby source code of an application was written in Java, and the Java classes which belonged to an application were compiled into dalvik byte-code (similar to java byte-code). Each .java file was compiled into a .dex file (similar to .class files) and these .dex files were combined together with the resources required by the application (like images...) and an Android application package (APK) was released for the user to install. During installation, further platform/hardware specific optimization was performed on the the .dex files and .odex files were produced. For execution, the Dalvik virtual machine (DVM) would load these .odex files into memory and execute the dalvik byte-code in them by interpreting them one-by-one. Each dalvik byte-code is provided an instruction handler. These instruction handlers are basically written in C and/or in assembly for
every architecture. Each instruction handlers is like an offset of a computer goto-like implementation with the byte-code being the selection mechanism. Depending on architecture, instruction-to-instruction transitions may be done as either computed goto or jump table. In the computed goto variant, each instruction handler is allocated a fixed-size area (e.g. 64 byte). "Overflow" code is tacked on to the end. In the jump table variant, all of the instructions handlers are contiguous and may be of any size. A Java function in invoked by the DVM (the run-time) via a function dvmCallMethod, which essentially pulls the byte-code corresponding to the particular method from the .odex file (which is loaded onto the memory) and begins interpreting them one-by-one. Figure 3.1 (Source: Wikipedia) shows a good graphical interpretation of the APK of the DVM.
3.2 What is the new Android run-time (ART)?

Interpreting methods tends to make a system significantly slower, to improve on performance Android made a big decision to adopt native code. For backward compatibility, the APK structure had to remain the same, hence none of the Java code could be compiled to native code during release time. This cannot done, also because the target architecture of an
application is unknown at release time, this data is only known during install time. During
the installation of the application, a tool called **dex2oat** is invoked which compiles every
single Java/dalvik method, one class after the other for all classes in the .dex file (present
in the APK) one-by-one into native code specific to the architecture of the device. After
compilation, its back-end **oatwriter**, combines the older .dex files with the compiled native
code and creates an OAT file. As we see in Figure 3.2 the OAT file is essentially an ELF
file (on the older DVM runtime, there was no executable code in an .odex file as all the
code was interpreted, so the whole file could be loaded onto memory as read/write, but the
.oat file requires a .text section which had to be executable, hence promoting an ELF
based file design). For backward compatibility and because of the constraint that some
dalvik byte-code just CANNOT be compiled into native code and HAVE to be interpreted,
the Android run-time (ART) had to still keep the .dex files with the original dalvik byte-
code. **Every Java method of a class can be either compiled to native code, also
known as ‘quick code‘ or not, in which case it is interpreted. Henceforth any
reference made to quick code or interpreted code refers to the Java functions
which were compiled accordingly.** Hence each class which is written to the .oat file
gets a label 1. kOatClassAllCompiled - All functions of the class is compiled into native
code. 2. kOatClassSomeCompiled - Some of the functions of the class are compiled to
native code. 3. kOatClassNoneCompiled - None of the functions are compiled to native
code and all of them have to be interpreted. **OatDexFile** is used to hold information about
a corresponding .dex file as well as to point to all the OatClasses which belong to the
particular file. **DexFile** is the exact same data structure used in the DVM based android,
more about this will be explained in the later sections in detail as and when required but
it is important to note that a .dex file is unaware of the presence of any native code and
is an independent entity (this is important because a user cannot have a reference to a
.dex file based data structure and derive the offset of the corresponding native code is a
straightforward manner, this will be dealt with in the later sections). **OatClass** is data
which can be reached via the OatDexFile and holds a list of offsets corresponding to the compiled methods of the particular class. Following this is space for other important runtime information like bitmaps for the Garbage Collector (GC), VmapTables which map the Virtual registers to memory addresses etc.. Following this is the quick/native code, which can only be referenced via the offset information in the OatClass, they contain a minimalistic header called the OatMethodHeader, which holds information like code_size, code_offset, gc_map_offset etc...

Figure 3.3 shows the memory dump of an OAT file, in this case the system@framework.oat file where we observe the layout as described in Figure 3.2. Now that we have a good overview of the file structure let us explore the run-time!
### ELF Header

Magic "0x7f ELF"

Simple ELF header with section information.

### OAT Header

Magic "OAT \n 039 \n 0"

Variable length with count of D OatDexFiles.

<table>
<thead>
<tr>
<th>OatDexFile[0]</th>
<th>OatDexFile[1]</th>
<th>...</th>
<th>OatDexFile[D]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One variable sized OatDexFile with offsets to Dex and OatClasses.

<table>
<thead>
<tr>
<th>DexFile[0]</th>
<th>DexFile[1]</th>
<th>...</th>
<th>DexFile[C]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One variable sized DexFile for each OatDexFile. These are literal copies of the input .dex files. Here exists the dalvik byte code corresponding to each .dex file.

<table>
<thead>
<tr>
<th>OatClass[0]</th>
<th>OatClass[1]</th>
<th>...</th>
<th>OatClass[C]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One variable sized OatClass for each of C DexFile::ClassDefs. Contains OatClass entries with class status, offsets to code, etc. This is important because it holds the offset into the native code for each method.

GcMaps, VmapTables, MappingTable and padding as following this will be the native code which needs to be aligned

<table>
<thead>
<tr>
<th>OatMethodHeader[0]</th>
<th>MethodCode[0] (native)</th>
<th>...</th>
<th>OatMethodHeader[N]</th>
<th>MethodCode[N] (native)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fixed size header for a CompiledMethod including the size of the MethodCode. One variable sized blob with the code of a CompiledMethod. Pairs are deduplicated.

Figure 3.2: OAT File Layout
Figure 3.3: Framework OAT file memory dump
3.3 libart.so: the heart of [ART] and its execution mechanism.

This section will serve as a bedrock for the later sections. libart.so, written in C++, is the main run-time library which handles and executes an android application. When the application starts up, via any entry point, libart.so starts off by locating the .oat file belonging to the process to be started, checks its sanity and loads it onto memory and starts executing the application. It spawns certain essential threads, each thread corresponding to a thread on the Java side. The main application starts thread as soon as it spawns, calls the Java function void java.lang.Thread.run() starts up and initializes all the required framework classes. Once this is done the <clinit> function belonging to the application is invoked, which initializes classes belonging to the application and starts executing the application. This is a view from the java perspective, but how does the libart.so handle function invocation?. As we discussed earlier, Java methods are compiled into quick code or interpreted code. Once the .oat file is parsed and loaded onto memory, it initializes and populates data structures in the run-time(libart.so) which mirror features on the Java side. By Java features we mean Thread, Objects, Classes, Java methods...are all mirrored on the run-time. To study method invocation behavior, of particular interest to us is the ArtMethod ((2)) class in libart.so which essentially mirrors a Java Method. Now, an instance of the ArtMethod might be either compiled into quick code, or may not have any quick code. An ArtMethod is invoked in [ART] runtime via a two key functions, void ArtMethod::Invoke(Thread* self, uint32_t* args, uint32_t args_size, JValue* result, const char* shorty), belonging to ArtMethod OR bool DoCall(ArtMethod* method, Thread* self, ShadowFrame& shadow_frame, const Instruction* inst, uint16_t inst_data, JValue* result). The former is used to handle functions compiled as quick-code/native-code, while the latter is used for interpreted code(called using handlers of invoke-XXX/range dalvik instructions). The
Thread data structure, which mirrors a java thread, holds a managed stack of all the invoked (java) methods belonging in it. ArtMethod::Invoke performs the following functions

1. performs some security checks to make sure there were no overflows etc ..

2. checks if the ArtMethod contains native code and if checks pass, invoke art_quick_invoke_stub

3. art_quick_invoke_stub is a architecture specific Assembly (ASM) method which sets up the stack frame and registers, extracts the method offset form the ArtMethod data structure and jumps to this offset. The Stack and registers for the method are laid out as shown in Figure [3.4]

DoCall on the other hand, is used to handle interpreted .dex methods and works by allocating and setting up a shadow stack for the target method and then, the dalvik byte-code corresponding to the ArtMethod, which is actually stored in a data structure called, Dex::CodeItem is extracted for the particular method and submitted to the interpreter functions, which interprets and executes each of the instruction one by one based the goto-implementation based method we spoke about earlier. A layout of Dex::CodeItem is shown in Figure [3.5]. This is a key data structure as it points us to the dalvik byte-code corresponding to the particular method.

3.4 Final words.

Now that we have a fair idea about the execution pattern of the Android run-time (ART), we can begin to model our dynamic analysis tool to support some of the features we require to perform unpacking.
Quick invocation stub internal.

On entry
r0 = method pointer
r1 = argument array or null for no argument methods
r2 = size of argument array in bytes
r3 = (managed) thread pointer
[sp] = JValue result
[sp + 8 ] = core register argument array
[sp + 12 ] = fp register argument array

uint32_t * fp_reg_args
uint32_t * core_reg_args
result_in_float
Jvalue* result

uint32_t out[n-1]
... ...
uint32_t out[0]
ArtMethod*

Figure 3.4: Quick invoke, stack and register layout

<table>
<thead>
<tr>
<th>registers_size_</th>
<th>ins_size_</th>
</tr>
</thead>
<tbody>
<tr>
<td>outs_size_</td>
<td>tries_size_</td>
</tr>
<tr>
<td>insns_size_in_code_units_</td>
<td></td>
</tr>
<tr>
<td>insns_</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: 32-bit code layout of the Dex::CodeItem data structure which holds the dalvik byte-code.
Chapter 4

DroidScope: ART

4.1 Overview and older Droidscope

DroidScope is a dynamic program analysis tool developed on top of the Android emulator which provides a user with a host of very useful Virtual machine introspection tools like basic-block tracing, native call tracing, dalvik call tracing, native instruction tracing, dalvik instruction tracing. ...DroidScope reconstructs both the OS-level and Java-level semantics simultaneously and accurately. It provides a platform for the user to dynamically load and unload plug-ins and a set of Application Program Interface APIs to be used in the plug-ins to which expose its various features. Basic block level callbacks are implemented by hooking key points in the in the Translated Block translation is the android emulator (for more information please refer to (18), (12), (9)). Native-level API callbacks are implemented on top of Basic-block callback (this will be explained in more detail in the following section). Dalvik method callbacks are implemented by hooking the dvmCallMethod function in the Dalvik virtual machine and reading data about the method like name, dalvik byte-code...from the Method data structure. Dalvik instruction callbacks are implemented by hooking a range of functions which correspond to the interpretation of all the dalvik byte-code, once execution reaches any of these functions, the contents of the DexPc is read
and disassembled. This older version of Droidscope supported android versions "4.2"- and only the Dalvik virtual machine [DVM].

4.2 The new Droidscope with ART support

To accommodate to a completely different run-time in ART we have had to comprehensively re-work DroidScope to still be able to accurately recover both Java and Native level semantic features. This section will provide an expansive overview of how we achieved this step-by-step and a lot of implementation detail will be based off of the details discussed in chapter 3.

4.2.1 Recovering Native semantics.

Below is a list of linux-level semantic information that Droidscope provides along with a very brief summary of how they are implemented. We will not dive deep into this as it is covered in detail in [13], [12], although a basic overview is imperative to understand the higher level working of the system. All or most of these features were ported from the older version of Droidscope with changes as and when required.

1. **Basic block entry/exit callback** - This is a very important concept and will be repeated several times while discussing the implementation. A basic block is defined as a block of ARM(this can be any architecture) instructions terminated by a jump or by a virtual CPU state change which the translator cannot deduce statically. A basic block begin/end is when code jumps into/out of a piece of code form another basic block. This is captured by hooking the translation procedure of QEMU([13], [12], [9]). This is a very powerful tool because a function begin is in essence a basic block begin, hence native functions tracing is built on top of basic block begin tracing as well as a host of other features.

2. **Process map** - A list of all processes running on the guest system and some information like PID, PGD, memory modules loaded...for each process. This information
is collected by hooking the `fork` system call (or a variant) of the Linux kernel and reading the kernel’s process linked list and looking for newly added processes.

3. **Memory-mapped modules for each process** - All code (+data) required for each process is loaded from the file-system onto memory to be executed. A process’s `vm_areaStruct` holds a linked list of all loaded modules for the process. We hook kernel functions which mutate this list, and at these hooks, detect new module loads, and collect information like (a) base address at which module is loaded (module_base_address) (b) size of the module (c) name of the module (d) inode_number corresponding to each module etc....

4. **Native function tracing** - On the new version of Droidscope we develop a new method of Native function tracing, whereby for each loaded native module, which is basically an ELF binary, or an ELF shared library (essentially the only two places where executable native code exists), as required, we use a file system forensics tool called "Sleuthkit" [7] to dump the file corresponding to the inode_number of the module into our host system, use a simple elf disassembler and collect offsets of all exported functions/symbols belonging to the module. Now, (module_base_address + offset) of a function is equal to the address of the function in memory and during a basic block begin callback, when the program counter is equal to (module_base_address + offset) for any function, then that basic block is the beginning of the particular native function. This is a very useful feature and will be used extensively in the sections to come.

5. Apart from these, Droidscope provides various other features like, native instruction begin/end callbacks, memory read/write callbacks, module load/unload callbacks, wrapper functions to read a blob of memory from the guest ram etc...
4.2.2 Recovering Java semantics: Introduction

Now that we have a good understanding of how native level semantics are recovered in Droidscope, let us discuss how we designed our system on top of these features to give us the same flexibility and features from a Java point-of-view for the new Android run-time (ART). Let us first take up process/java application creation callbacks. Java applications, like native applications are spawned by `forking`, but they differ in that the application name is not resolved when a fork takes places, hence we employ a different approach to track the creation of these processes by 1. making a list of all processes which have been `forked` by an application called `main`, which is the `init` process for every android application. All processes on this list are not android application 2. we hook a function in the `libcutils.so` library called `set_process_name` which is used to set the name of a particular android application. In this hook function we check if the process currently calling `set_process_name` is on our list, if so, we upgrade the process data structure with the name of the application and invoke our android application process begin callback with the accurate name of the application. This is an important tool as a user can now begin tracing of a program by its name.

Next task is recovering java function level semantics, or a Java function callback. During the installation of an application, as discussed before, a compiler/tool `dex2oat` is invoked on the [APK](#) to compile it into an [ART](#) specific [OatFile](#), and one of the arguments that can be passed to this compiler is `dalvik.vm.dex2oat-filter`, the options for which are either "speed" or "interpret-only", the former instructs the compiler to translate AS MANY functions as possible into native code leaving the others as interpreted functions, and the latter, commonly used for debugging instructs the compiled to not compile ANY function to native code. By default this argument is set to "speed". For all intents and purposes, from now on we will consider function tracing as obtaining a trace/`names` of every single Java function executed by an single application. No packing behavior is expected from the application and that problem will be dealt with after this section, building upon this. Since there is not really one single function
on the Android run-time \textsc{ART}, as we saw in the previous chapter, we need to come up with a more sophisticated technique to handle both quick code or compiled methods and interpreted methods, in separate independent methods. Let us look at an overview of each implementation following the general algorithm used.

### 4.2.3 Compiled \textsc{ART} Methods

Referring to Figure 3.2 we see that the compiled native methods corresponding to Java methods are present in the very end of the oatfile, and observe that they are laid out very similar to how native functions would be laid out in an executable or native shared library. Hence to trace there methods, we can take a similar approach as tracing native methods, by collecting the offsets of all the compiled methods and corresponding each offset to a Java method (we can collect the name for reference). For native binaries/shared libraries, as their formats are pretty simple, extracting offsets from them is a fairly easy process BUT for oatfiles, we need to follow a different process. The general idea is to obtain the the oatfile belonging to the method, and extract offsets from it.

We know from our discussion in chapter 3 that the function, \texttt{void ArtMethod::Invoke(Thread* self, uint32_t* args, uint32_t args\_size, JValue* result, const char* shorty)} from \texttt{libart.so} is usually used to dispatch a ‘compiled method’, this by no means covers all compiled methods, but it is sufficient for us to use as a hook point to perform offset extraction from the OatFile belonging to the called ArtMethod. Then next challenge is to discover which module the called ArtMethod belongs, because as we recall the function call is in \texttt{libart.so} and ArtMethod is a very simplistic data structure which does not contain any back reference to the originating module. We can reach the DexFile data structure from the ArtMethod data structure and this method will be used for both compiled and native code extraction. Let us discuss that and come back to the offset extraction problem.
DexFile data structure from ArtMethod

It is important to note that because we are using a dynamic analysis platform, and performing analysis in a plug-in running on the host system and profiling the guest android system, all pointers to these data structures point to memory on the guest RAM and not the host RAM. They cannot be dereferenced directly nor can be used to invoke any class functions, to read field data from a pointer to a data structure in the guest memory, we first need to read a block of memory starting from the pointer up to a size equal to the size of the data structure. We can then cast this block to a pointer to the original data structure and read data from it, BUT caution is be taken as only POD - based data structure can be read with this method, if there are more pointers in the data structure, this process has to be repeated. A sample of this process can be seen in Code Listing 4.1.

Listing 4.1: Sample code showing how guest data structures are read from memory

```c
// The ArtMethod is read off of r0
target_ulong called_art_method = env->regs[0];

// Get the ArtMethod from guest memory.
art::mirror::ArtMethod *methodzz;
char block1[SIZEOF_TYPE(art::mirror::ArtMethod)];
DECAF_read_mem_with_pgd(env, pgd_strip(cr3), called_art_method, block1,
             SIZEOF_TYPE(art::mirror::ArtMethod));
methodzz = (art::mirror::ArtMethod *)(block1)
```

Back to the problem at hand, our goal is to find out the module to which this particular ArtMethod belongs to so that we can perform offset extraction from this module, and as the ArtMethod itself does not hold any clue to this, we aim to extract the DexFile Data structure, which holds the base address of the DexFile, and as Figure 3.2 shows us, if we find out which module the DexFile belongs to, and since the DexFile is embedded inside the OAT file, we find the module loaded into memory corresponding to the OatFile. To move
from the ArtMethod to the DexFile we follow the following steps

1. The ArtMethod data structure has a field called
   
   HeapReference<Class> declaring_class_; which is a pointer to the mirror data
structure of the class to which this method belongs to. HeapReference is just a
wrapper around the pointer to the class which is technically on the Java heap. As this
data structure is laid out in the class memory completely as a value type, we use the
same method as shown in Code Listing 4.1 to first read the ArtMethod from the guest
memory and then read the class pointer.

2. Once we have a pointer to the class data structure which was wrapped inside the
   HeapReference, this too does not contain a direct reference to the DexFile to which
the class belongs to but it contains a DexCache data structure which basically holds
resolved copies of strings, fields, methods, and classes from the dexfile as well as a
pointer to the DexFile data structure to which this class belongs to (Voila!). We
extract the DexCache and finally the pointer to the DexFile using similar methods
explained above.

3. Once we have the DexFile pointer, we dump and read the whole DexFile data struc-
ture. In benign circumstances, the DexFile data structure is a treasure which contains
a lot of important data required for Virtual machine introspection (VMI), but right
now we are only concerned with const byte* const begin_; which points, in the
guest memory, where the DexFile starts.

Offset extraction from the OatFile

As mentioned earlier, DroidScope provides an API which, given a particular process and an
address, returns the module to which this address belongs to. Once we have this information,
we dump the OatFile from memory completely and extract the offsets based on Algorithm 1.
This is a fairly complicated and intricate process were we iterate through essentially each of
the DexFiles present inside the OatFile and then walk each class of every DexFile. Then for each class, we do sanity checks to verify the class and finally for each method (OatMethod), which contains compiled code, we extract it along with the name of the function and add it to our internal map as seen in Line 30 in Algorithm [] We internally maintain a data structure such as

\[
\text{MAP}[\text{module\_base} -> \text{MAP}[\text{code\_offset} -> \text{method\_name}]]
\]

**Final Java function tracing for native ART methods.**

This works well because performing the particular parsing every single time would be highly time consuming and to solve that every module’s OatFile offsets are extracted once. Now that we have all the offsets of compiled code in an OatFile we can now trace native code execution for the particular application to retrieve Java level function call tracing. To achieve this we register for a Basic Block begin callback, in which we check first the current module to which this Basic Block belongs to, then if this module is an OatFile and has offsets extracted for it, we check if the current Program Counter is equal to \{ module\_base + offset \} for any of the extracted offsets, if so then this is the start of a Java function pointed to by the method\_name corresponding to the offset in the MAP.

### 4.2.4 Interpreted Java methods

Depending on certain options, a large/small part of an android [APK] is still interpreted. This means that tracing Java functions should include tracing of dalvik interpreted methods. As discussed in Chapter [chapter 3], interpreted methods are ALL dispatched via a function call, bool DoCall(ArtMethod* method, Thread* self, ShadowFrame& shadow_frame, const Instruction* inst, uint16_t inst_data, JValue* result) in the Android run-time [ART]. Simply hooking this function and employing a simple process to extract the name of the method from the ArtMethod will be sufficient.
Algorithm 1 Extract offsets of each compiled method from the OatFile

1: procedure EXTRACT_ART_OFFSETS(module_base, module_size, module_name, process_identifier)
2:     if module_base == extracted then
3:         Return
4:     end if
5:     oat_file_contents ← ReadGuestMemory(process_identifier, module_base, module_size)
6:     oat_file_valid ← CheckOatMagic(oat_file_contents)
7:     if oat_file_valid == false then
8:         Return
9:     end if
10:     host_oat_file_dump ← DumpContentsToFile(oat_file_contents, module_name)
11:     oat_file ← ARTOatFileOpenMemory(oat_file_contents, host_oat_file_dump)
12:     if OatFile == nullptr then
13:         Return
14:     end if
15:     oat_dex_files ← oat_file->GetOatDexFiles()
16:     for each oat_dex_file in oat_dex_files do
17:         dex_file ← oat_dex_file->OpenDexFile()
18:         class_defs ← dex_file->GetClassDefs()
19:         for each class_def in class_defs do
20:             oat_class ← oat_dex_file->GetOatClass(class_def)
21:             class_data ← dex_file->GetClassData(class_def)
22:             if class_data != nullptr then
23:                 oat_methods ← oat_class->GetOatMethods()
24:                 for each oat_method in oat_methods do
25:                     method_name ← PrettyMethod(oat_method)
26:                     code_offset ← oat_method->GetCodeOffset()
27:                     AddOffsetAndNameToMap(module_base, method_name, code_offset)
28:                 end for
29:             end if
30:         end for
31:     end for
32: end procedure
Java Function name from ArtMethod

Since just hooking the above mentioned function will give us full coverage of all interpreted functions, we can go ahead by extracting the function name. To do so we can start with the first argument, the ArtMethod. In an android APK the DexFile present inside the OatFile is the only location where strings like method names, class names, parameter list... are present. The DexFile contains a field MethodId for all methods present in the file, this contains offsets into the DexFile to other important data structures like the class to which the method belongs to, the name of the method.... The DexFile also contains an array of StringId data structure for all strings present in the file, were each StringId essentially contains the offset to the particular string from the beginning of the DexFile and any other data structure in the DexFile, which needs to point to a string, hold a reference into this array. Hence to extract the name we perform the following steps

1. Extract the DexFile from the ArtMethod as described in section 4.2.3.

2. Read the dex_method_index_ field from the ArtMethod which is essentially an index into the MethodId array pointing to the MethodId for this particular method.

3. Read that particular MethodId from the DexFile and extract the name_idx_ field in it, which is essentially an index into the StringId array pointing to the name of this particular method.

4. Read the particular StringId, which contains a field string_data_off_ which is the offset of a leb128 encoded string from the beginning of the DexFile

5. Decode and read the string present at the address (dex_file_begin_ + string_data_off_) and print/dump out the function name.
4.2.5 Recovering Java semantics: Final Picture

Now, in this section we combine the algorithms from subsection 4.2.3 and subsection 4.2.4 into a single algorithm and showcase its implantation on Droidscope. Algorithm 2 summarizes the process of java api tracing by covering both interpreted and native java functions in an abstract view. The process is implemented in a basic-block (BB) begin callback (CB) function in Droidscope:ART. As discussed in subsection 4.2.1, a basic-block (BB) is defined as a block of ARM (this can be any architecture) instructions terminated by a jump or by a virtual CPU state change which the translator cannot deduce statically and a basic block begin/end is when code jumps into/out of a piece of block of code from another basic block. We begin in the callback to check if the CB belongs to the process being tracked, if so we extract the native function call correspond to this BB (only if the BB is the first one of a function then that is a function call). If this a call to doCall... or ArtMethod::Invoke, the two functions that we have talked about, then we follow the process described in section 4.2.3 to extract the DexFile from the ArtMethod. Once we have the DexFile. If the function call was to ArtMethod::Invoke, then we go ahead and extract all the compiled method offsets from the particular DexFile's OatFile. If the function call was to doCall... then we go ahead and extract the function name for the particular ArtMethod with the process as described in section 4.2.4 and dump it.

After this is done, we check if the the current module, to which this basic block belongs is an OatFile of which he have extracted offsets for. If so, we check if the current program counter is equal to the sum of the module’s base address plus one of these offsets. If so, we extract the function name corresponding to the offset and dump this.
Algorithm 2 JAVA API Call tracer for Droidscope : ART

1: procedure block_begin_cb(cpu_state_information)
2:   current_pc ← GetCurrentPc(cpu_state_information)
3:   current_cr3 ← GetCurrentCr3(cpu_state_information)
4:   if (current_cr3! = tracked_cr3) then
5:     Return
6:   end if
7:   current_module ← GetCurrentModule(cpu_state_information)
8:   if (current_module == "libart.so") then
9:     current_function ← GetCurrentFunction(cpu_state_information)
10:    if (current_function == ("doCall..."||"ArtMethod :: Invoke...") then
11:       dex_file ← GetDexFileFromArtMethod() \(\text{\footnotesize section 4.2.3}\)
12:       dex_file_module ← GetModuleForAddress(dex_file.begin_)
13:       if (current_function == "ArtMethod :: Invoke...") then
14:         EXTRACT_ART_OFFSETS__(dex_file_module)[Function -1]
15:         Return
16:       else
17:         dex_method_name ← GetDexMethodName(dex_file, art_method) \(\text{\footnotesize section 4.2.4}\)
18:         DUMP_FUNCTION_NAME(dex_method_name)
19:         Return
20:     end if
21:   end if
22:   if (ArtOffsetsForModuleBase(dex_file.begin_) == TRUE) then
23:     dex_method_name ← GetDexMethodNameForOffset(current_pc)
24:     DUMP_FUNCTION_NAME(dex_method_name)
25:     Return
26:   end if
27: end procedure
Chapter 5

DroidUnpack

Now that we have a solid understanding of the execution environment on the Android runtime (ART) and a reliable analysis engine which is able to get a complete execution trace of Java function calls made by an application, we can build our unpacker on top of it. Before we proceed we present a series of objectives that we want to be able to achieve with this unpacker.

5.1 Objectives

1. A generic unpacker which is able to dump/log the code execution of every single Java function for a packed android application regardless of features used by a packer.

2. Furthermore, the unpacker should be able to log all memory writes made by the application and automatically detect if any of these regions in memory was executed.

5.2 Code Extraction

Although we were able to implement Java function call tracing successfully, special care must be taken to track the execution of packed applications. Runtime packers try very hard to obfuscate their true intentions and make it very hard for a security analyst to
study execution flow of the application. Additional to this we need to extract code, dalvik byte-code for interpreted method and arm instructions for native ART methods as function names are sometimes simply obfuscated/mangled by a packer and hence simply extracting function name is insufficient for a security analyst to perform further evaluation. What data structures to read and how to do so from memory should be carefully chosen because runtime packers sometimes perform selective unpacking, header mangling etc which means that the extraction technique must be immune to these features. The following approach is used for native and interpreted method,

1. Native methods: One of the challenges of the new Android run-time (ART) was to study the execution of native code in a Java context. Every time we detect the beginning of a compiled ART function, because we have extracted the offsets beforehand from the oatFile we have also collected the size in memory of these methods. We go ahead and dump the native code from the start to its end. For good analysis it is preferred to have dalvik byte code for all functions rather than native code to keep things consistent, but this is a harder challenge because strict checks are performed on the ART native code, but no strict checks are performed on the byte code during installation. Hence a packer can, after installation wipe out all the dalvik byte-code from the dex files in memory, because they are not required for the execution of the application. The only option to make the most generic would be to disassemble the native code into byte-code, this is a particularly challenging problem and is under our future plans.

2. Interpreted functions: Interpreted methods are comparatively easier in that, each interpreted functions is assigned a data structure in the Android run-time (ART), called DexFile::CodeItem, shown in Listing 5.1. Similar to extracting the name of the function for interpreted functions, the ArtMethod data structure holds an reference its CodeItem data structure present in its DexFile. We proceed by extracting this data structure and disassembling/logging insns_size_in_code_units_ number of
instructions starting from `insns_` as seen in `CodeItem`.

Listing 5.1: DexFile::CodeItem

```c
// Raw code_item.
struct CodeItem {
    uint16_t registers_size_;  
    uint16_t ins_size_;        
    uint16_t outs_size_;       
    uint16_t tries_size_;      
    uint32_t debug_info_off_;  
        // file offset to debug info stream
    uint32_t insns_size_in_code_units_;  
        // size of the insns array, in 2 byte code units
    uint16_t insns_[1];        
};
```

5.3 Unpacking algorithm and other features.

The unpacker is essentially a plugin on `Droidscopec:ART` which is developed on top of the Java API tracer plugin we described previously with some additional features added. With very little addition to the JAVA API tracer, we are able to implement a generic, robust unpacker for the latest Android run-time [ART] as seen in Algorithm-3. We insert an additional memory write callback, which essentially records all memory writes made by an application. Since these packers tend to dynamically write into the target `OatFile` to correct portions of the code before execution, we can study such behavior with the help of this callback. Apart from this the only additional features is the code extraction which is performed for each method, as described in section 5.2.
Algorithm 3 DroidUnpack plugin for Droidscope:ART

**procedure** MEMORY_WRITE_CALLBACK(virtual_address, cpu_state_information)

  current_cr3 ← GetCurrentCr3(cpu_state_information)

  if (current_cr3 != tracked_cr3) then
    Return
  end if

  RECORD_ADDRESS_WRITE(virtual_address)

**end procedure**

**procedure** BLOCK_BEGIN_CB(cpu_state_information)

  current_pc ← GetCurrentPc(cpu_state_information)
  current_cr3 ← GetCurrentCr3(cpu_state_information)

  if (current_cr3 != tracked_cr3) then
    Return
  end if

  current_module ← GetCurrentModule(cpu_state_information)

  if (current_module == "libart.so") then

    current_function ← GetCurrentFunction(cpu_state_information)

    if (current_function == ("doCall..."||"ArtMethod :: Invoke...") then

      dex_file ← GetDexFileFromArtMethod() [section 4.2.3]
      dex_file_module ← GetModuleForAddress(dex_file.begin_)

      if (current_function == "ArtMethod :: Invoke...") then

        if (art_offsets_extracted(dex_file_module) == TRUE) OR
           (dirty_memory_write_in_module(dex_file_module) == TRUE) then
          EXTRACT_ART_OFFSETS__(dex_file_module)[Function -1]
        end if
      else
        dex_method_name ← GetDexMethodName(dex_file, art_method) [section 4.2.4]
        DUMP_FUNCTION_NAME(dex_method_name)
        dex_code_item ← GetDexCodeItem(dex_file, art_method)
        DUMP_CODE_CODE(dex_code_item)
      end if
    end if
  end if

**end procedure**
if (ArtOffsetsForModuleBase(dex_file.begin_) == TRUE) then
  dex_method_name ← GetDexNativeMethodNameForOffset(current_pc)
  dex_method_size ← GetDexNativeMethodSizeForOffset(current_pc)
  dump_function_name(dex_method_name)
  if (DirtyMemoryInRange(current_pc, current_pc + dex_method_size) == TRUE) then
    dump_dirty_code_metadata()
    end if
  end if
end if
end procedure
Chapter 6

Results

6.1 Overall results

We conduct a simple evaluation to measure the veracity of our work, by running the same unpacker plugin, unchanged on 10 applications each packed by 6 standard commercial packers, "Alibaba Inc." (1), "Qihoo360 Inc." (6), "Tencent Inc." (8), "Bangcle Inc." (4), "Baidu Inc." (3) and "Ijiami Inc." (5), as well as on the unpacked application. Verifications was done manually on the results (function call and code dumps) comparing the result of each packer with the that from the unpacked application. We were successfully able to extract code from 5 of the 6 packers we tested with for all applications except for the Qihoo360 Inc packer which incorporates an anti-emulation feature. The results are seen in Table 6.1. We compare each of the packer with data extracted from the original unpacked version of the application present 3 parameters, Code unpacked %(CU), which is simply amount of original function flow we could extract from the packed samples, Dirty Native Code %, which was the percentage of compiled native java functions which were dynamically mutated by the application during the runtime of the application and finally Dirty Dalvik Code %, a similar parameter which indicates the percentage of interpreted Dalvik functions which were dynamically mutated. In the next section we take a look at all these packers
<table>
<thead>
<tr>
<th>Applications</th>
<th>Alibaba Inc</th>
<th>Baidu Inc</th>
<th>Bangke Inc</th>
<th>Ijiami Inc</th>
<th>Qiho360 Inc</th>
<th>Tencent Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.banasiak.coinflip</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [89.41]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [70.37]</td>
<td>✓ [13]</td>
<td>✓ [3.57]</td>
<td>X</td>
</tr>
<tr>
<td>com.github.sanbeg.flashlight</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [89.47]</td>
<td>✓ [2.08]</td>
<td>✓ [9.52]</td>
<td>X</td>
</tr>
<tr>
<td>com.frankcalise.h2droid</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [90]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [16.98]</td>
<td>✓ [8.51]</td>
<td>✓ [3.64]</td>
<td>X</td>
</tr>
<tr>
<td>com.tortuca.holoken</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [85.33]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [65.06]</td>
<td>✓ [10.86]</td>
<td>✓ [4.64]</td>
<td>✓ [4.17]</td>
</tr>
<tr>
<td>ru.gelin.android.browser.open</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [3.8]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [65.06]</td>
<td>✓ [10.86]</td>
<td>✓ [4.64]</td>
<td>✓ [4.17]</td>
</tr>
<tr>
<td>edu.killerud.fileexplorer</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [13.63]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [55.55]</td>
<td>✓ [3.59]</td>
<td>✓ [18.18]</td>
<td>X</td>
</tr>
<tr>
<td>edu.killerud</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [15]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [71.42]</td>
<td>✓ [1.7]</td>
<td>✓ [22.22]</td>
<td>X</td>
</tr>
<tr>
<td>de.boesling.hydromemo</td>
<td>CU %</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DNC [%]</td>
<td>X</td>
<td>X</td>
<td>✓ [97.36]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DDC [%]</td>
<td>✓ [100]</td>
<td>✓ [31.25]</td>
<td>✓ [7.45]</td>
<td>✓ [11.11]</td>
<td>X</td>
</tr>
</tbody>
</table>

individually, for one of the applications, **com.banasiak.coinflip**.

---

1. Code Unpacked %.
2. Dirty Native Code [%]
3. Dirty Dalvik Code [%]
6.2 Case Studies

6.2.1 Alibaba Inc.

This packer adds two shared libraries, `libmobise.so` and `libmobisecx.so` into the APK. The original `.dex` file is packed and encrypted and its replaced by a standard custom made `.dex` file which acts as a launchpad to bring up their shared library, `libmobise.so`, which performs bulk of the unpacking. This library itself is obfuscated and it corrects itself, extracts the the original `.dex` file and loads the it into the shared library `libmobisecx.so`, which is under its control which can be observed in Figure 6.1. We see that three ART/OAT files were loaded, the first one being the `framework` file, the second being their launchpad file and the third the actual application. We can verify this is the java function traces.

We see in Figure 6.2 the comparison of the java api trace between the Alibaba packer and the unpacked application. We see that we have been able to successfully get the same functions flow as the unpacked application even with this application being packed. Also, it
is interesting to see that every function call made to the original .dex file is followed and preceded by custom function calls. The packer has inserted these functions calls to inject a form of indirection which would render many analysis techniques fruitless. Also interesting to note is that as we observed before, the .dex file is actually embedded in their own shared library. This is a clever strategy as it gives the packer full control of the read/write into the code, but as we discussed in the our motivation section, we rely on the fact that the for the application to execute naturally, it has to adhere to certain constraints imposed by the Android run-time (ART), meaning that some data structures in the run-time can never be mutated and we exploit this to unpack there applications.
Figure 6.2: Comparing the Java API traces between the unpacked and the packed application output both obtained from our unpacker plugin.
6.2.2 Baidu Inc.

Similar to the previous packer, Baidu includes a custom shared library libbaiduprotect.so which performs the similar role of decrypting/unloading and starting the application. Here as seen in Figure 6.3, the packer spawn another child process where the actual application is executed. We also observe that the framework, the launchpad and the actual ART/OAT files were successfully loaded and the target file is actually present in the heap. Let us now have a look at the java api calls.

Figure 6.3: Behavior of the Baidu packer.

As we observe the java api calls in the compare them with the the unpacked application, in Figure 6.4 we see that again we are able to successfully unpack the application. It is interesting to note here that the target application is actually loaded somewhere on the heap/dynamically allocated memory.
Figure 6.4: Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin.
6.2.3 Bangcle Inc.

Bangcle was one of the more sophisticated packers, with more packing features employed than others. The shared library used was `libsecexe.so` which performed the unpacking. The actual code was contained in a `classes.dex` file which was dynamically recovered and loaded into memory to start the application. The packer employed several child processes, one of them to observe for p-trace based debugging detection and the other where the application was actually loaded. The interesting part of this packer is that it employed another `.dex` file, called `container.dex` which actually performed runtime unpacking of the application as it executed (more on this in the next paragraph). Also interesting to note was that unlike other packers that we studied, Bangcle’s target/actual application had all of its functions compiled into native ART functions. We see this behavior in Figure 6.5.

Now let’s take a look at some of the more interesting features employed by this packer.

Figure 6.5: Behavior of the Bangcle packer.
Some interesting features

The most interesting feature about the Bangcle packer is the container.dex. This is actually an application launched by the parent application (most likely from the shared library) and acts as an ACTUAL container to the original application. What this means is that all data (like strings, classes, resources...) are encrypted when the application starts up and when there is a request to any of these elements, they are trapped by the container.dex application which decrypts the required data at runtime and provides it to the application. A more longer version of the api behavior of Bangcle can be found in Appendix for an interested user. Here in the shorter version in Figure 6.6 where we see that despite these efforts by the packer we are able to successfully unpack the application.

6.3 Ijiami Inc.

Similar to the other packers, Ijiami unloads its packed OAT file onto a file named ".1", as seen in Figure 6.7 compiles this application (we see the invocation to dex2oat) and after the application starts, deletes the backing file making it difficult for a security analyst to unpack the application. In Figure 6.8 we see that we are successfully able to recover the accurate execution trace of the application compared to its unpacked application.
(a) Java function call trace from the unpacked application.

(b) Java function call trace from the application packed with Bangcle.

Figure 6.6: Comparing the Java API traces between the unpacked and the packed application output both obtained from our unpacker plugin.
Figure 6.7: Behavior of the Ijiami packer.
(a) JAVA function call trace from the unpacked application.

(b) JAVA function call trace from the application packed with Ijiami.

Figure 6.8: Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin.
6.4 Qihoo360 Inc.

This unpacker employed an emulator detector which crashes the application on the android emulator. We see in Figure 6.9 that the application actually starts, invokes some shell commands and crashes unexpectedly.

![Figure 6.9: Behavior of the Qihoo360 packer.](image)

6.5 Tencent Inc.

Behavior was similar to other packers, the target ART/OAT file is loaded onto the dynamic heap and the executed as seen in Figure 6.10 and we see in Figure 6.11 successful code extraction.
Figure 6.10: Behavior of the Tencent packer.
Figure 6.11: Comparing the JAVA api traces between the unpacked and the packed application output both obtained from our unpacker plugin.
Chapter 7

Conclusion and Future work

7.1 Conclusion

In the years to come, the android execution environment will shift more towards the design of the new Android run-time \(\text{ART}\) to keep pace with other platforms like the iOS who’s complete execution is done as native code. In this project, Droidunpack we essentially provide a strong dynamic analysis framework, based on the new Android run-time \(\text{ART}\) to set the stage for performing research projects on the framework. In the process of doing so we study the \(\text{ART}\) in an exhaustive way in all its relevant aspects and present them for an interested security analyst. We then summarize the gist of important features required for us to implement a platform which will provide us with various Virtual machine introspection (VMI) features. Additionally, we are precisely able to recover both native and Java level semantics for Android applications on the new Android run-time \(\text{ART}\).

We lay out the necessary foundations required for the construction of a generic android unpacker. We briefly describe some of the work that has been done on unpacking android applications, their strengths and shortcomings. Acknowledging these factors, we look at the problem from a new perspective. We argue that for any packed android application, with the core of its original application written in java, regardless of what packer was used, has
to conform to some strict rules imposed by the Android run-time (ART), in that certain
data structures in the run-time are beyond the control of a packer. With this, we design our
unpacker by tapping into certain key events in the run-time of an application to successfully
perform unpacking of the application. We then evaluate the unpacker against some of the
standard commercial packers to verify the working and accuracy of the unpacker.

7.2 Drawbacks

Although we provide a robust and generic unpacking framework, it suffers from some small
issues like code-coverage and anti-emulation detection features. We unpack and extract all
the code executed by the application, with 'executed' being the key word. Since android is
a GUI based environment, simple testing where we start the application, trace it for some
fixed time and kill it will cover lesser code than what is actually present. This means that in
essence we are not able to cover and unpack the source code of the whole application. This is
a particularly difficult problem since packers employ features where they unpack a function
right before it is dispatched and decrypt it again after it finishes execution. In such a case,
it becomes a lot more difficult to generically unpack the complete application. To address
this we present some interesting plans that we mean to implement. Anti-emulation is a bane
for dynamic analysis tools based on emulators. Although there are no plain solutions to this
problem, there are always anti-anti-emulation tools to help avoid these to an extent.

7.2.1 Breaking out of DroidUnpack

One other important topic to think about is how attackers can subvert such an unpacking
mechanism. Apart from having anti-emulation stubs installed, an attacker can carefully
move the original implementation of the application, into native JNI code. Although a
sophisticated process, the user can find a way to convert the java/dalvik code present in the
APK into a custom representation and move it into shared native libraries, hence breaking
out of the the **ART** runtime.

### 7.3 Future Work

As next steps in the evaluation of this project we plan to perform more large scale analysis on a huge data-set of malicious/benign Android applications in the wild, to better understand a trend in their design and behavior. To address the problem of code coverage, we plan to design an engine for force execution of java functions combined with the unpacker.
Appendix A

Source code excerpts from the DroidUnpack plug-in.

Below is a small cut-down version of the unpacker plug-in. Code Listing A.1 basically provides the implementation of the different algorithms discussed in the document. Only the important callback functions are listed and other parts of the code are omitted to keep it compact.

Listing A.1: Source code from the DroidUnpack plug-in.

```cpp
/*
 * Copyright (C) <2012> <Syracuse System Security (Sycure) Lab>
 * This is a plugin of DECAF. You can redistribute and modify it under the terms of BSD license but it is made available WITHOUT ANY WARRANTY. See the top-level COPYING file for more details.

For more information about DECAF and other softwares, see our web site at:
http://sycurelab.ecs.syr.edu/

If you have any questions about DECAF, please post it on http://code.google.com/p/decaf-platform/
*/
/**
 * @author Abhishek VB
 * @date June 22 2015
 */

static void SkipAllFields(art::ClassDataItemIterator & it) {
    while (it.HasNextStaticField()) {
```

54
// Main algorithm which extract offsets of native functions from an OAT file.

static void extract_art_offsets_(target_ulong base_,
    target_ulong size,
    std::string name,
    CPUArchState* env,
    target_ulong cr3) {
    if (!base_to_offsets.count(base_)) {
        return;
    }

    // Try to grab the memory and open and OAT file
    std::vector<uint8_t> oat_file_contents;
    target_ulong oat_file_end = base_ + size;

    // For the range of module, read it from memory onto a buffer.
    for (target_ulong oat_file_base = base_; oat_file_base != oat_file_end;
        oat_file_base += 1) {
        uint8_t ph = 0;

        DECAF_read_mem_with_pgd(env, cr3, oat_file_base, (void**)&ph,
            sizeof(uint8_t));

        oat_file_contents.push_back(ph);
    }
    std::string name1 = dex_files_dir + std::to_string(current Dex_file) + ".oat";
    std::string calc_dump = name1;

    // Save the contents as a local file for analysis later.
    binary_save(oat_file_contents, calc_dump);

    // Skip through the ELF header.
    std::vector<uint8_t> elf_magic_needle{ 'E', 'L', 'F', '\'0' };
    std::vector<uint8_t>::iterator itt =
        std::search(oat_file_contents.begin(), oat_file_contents.end(),
            elf_magic_needle.begin(), elf_magic_needle.end());

    if (itt != oat_file_contents.end()) {
        oat_file_contents.erase(oat_file_contents.begin(), itt);
    }
    else {
        // No ELF header, return.
        return;
    }

    ++current Dex_file;

    // Skip over the OAT header.
```cpp
std::vector<uint8_t> oat_magic_needle{'0', 'a', 't', '
', '0', '3', '0', '0'};

std::vector<uint8_t>::iterator it =
    std::search(oat_file_contents.begin(), oat_file_contents.end(),
                oat_magic_needle.begin(), oat_magic_needle.end());

oat_file_contents.erase(oat_file_contents.begin(), it);

std::vector<uint8_t>::iterator it1 =
    std::search(oat_file_contents.begin(), oat_file_contents.end(),
                oat_magic_needle.begin(), oat_magic_needle.end());

if (it1 != oat_file_contents.end()) {
    oat_file_contents.erase(oat_file_contents.begin(), it1);
}

std::string error_msg;
std::unique_ptr<art::OatFile> oat_file{
    art::OatFile::OpenMemory(oat_file_contents, calc_dump, &error_msg)};
// CHECK(oat_file.get() != NULL) <= calc_dump <= ": " <= error_msg;

if (oat_file.get() == nullptr) {
    if (bad dex file bases.count(base_))
        bad dex file bases[base_]++;
    else
        bad dex file bases[base_] = 1;

    // Ugly!
    auto j3 = json::parse(get_string(json_path));
    j3["dex file integrity"] = false;
    std::string s = j3.dump();
    save_string(s, json_path);
    return;
}

monitor_printf(default_mon, "art file done 
", name.c_str(),
               name1.c_str());

std::unordered_map<target_ulong, std::string> to_add_offsets;
std::unordered_map<target_ulong, target_ulong> to_add_sizes;
const std::vector<const art::OatFile::OatDexFile*>& oat_dex_files =
    oat_file->GetOatDexFiles();

for (size_t i = 0; i < oat_dex_files.size(); i++) {
    const art::OatFile::OatDexFile* oat_dex_file = oat_dex_files[i];
    CHECK(oat_dex_file != nullptr);
    std::string error_msg;

    const art:: DexFile* dex_file = oat_dex_file->OpenFile(&error_msg);

    if (dex_file == nullptr) {
        std::cout <= "Failed to open dex file 
" << oat_dex_file->GetDexFileLocation() <= ":\n" <= error_msg;
        continue;
    }
    //...
for (size_t class_def_index = 0; class_def_index < dex_file->NumClassDefs();
    class_def_index++) {
    const art:: DexFile:: ClassDef& class_def =
        dex_file->GetClassDef(class_def_index);
    const art:: OatFile:: OatClass oat_class =
        oat_dex_file->GetOatClass(class_def_index);
    const byte* class_data = dex_file->GetClassData(class_def);
    if (class_data != nullptr) {
        art:: ClassData:: Iterator it(dex_file, class_data);
        SkipAllFields(it);
        uint32_t class_method_index = 0;
        while (it.HasNextDirectMethod()) {
            const art:: oatFile:: oatMethod oat_method =
                oat_class.GetOatMethod(class_method_index++);
            uint32_t code_offset = oat_method.GetCodeOffset();
            to_add_offsets[code_offset] =
                PrettyMethod(it.GetMemberIndex(), dex_file, true);
            to_add_sizes[code_offset] = oat_method.GetQuickCodeSize();
            it.Next();
        }
        while (it.HasNextVirtualMethod()) {
            const art:: oatFile:: oatMethod oat_method =
                oat_class.GetOatMethod(class_method_index++);
            uint32_t code_offset = oat_method.GetCodeOffset();
            to_add_offsets[code_offset] =
                PrettyMethod(it.GetMemberIndex(), dex_file, true);
            to_add_sizes[code_offset] = oat_method.GetQuickCodeSize();
            it.Next();
        }
    }
    base_to_sizes[base] = std::move(to_add_sizes);
    base_to_offsets[base] = std::move(to_add_offsets);
    base_to_oat_file[base] = (void*)oat_file.release();
}

// This is the memory write callback which registers writes made to memory
target_ulong current_cr3 = 0x00000000UL;
CPUArchState* current_cpu = NULL;
static void hook_writes(Decaf_Callback_Params* params) {
    if (!target_cr3s.count(current_cr3))
        return;

    byte_addrs_written.insert(params->mvw.vaddr);
}

// This is the heart of the unpacker, as described in the document, a basic-block
// is the piece of code which is terminated by a control-flow transfer instruction
// We hook at the beginning of each basic block and perform required extraction
// for each basic block.
static void block_begin_cb(Decaf_Callback_Params* param) {

char modname[1024];
char functionname[1024];
// char process_name[1024];

CPUArchState* env = param->bb.env;
target_ulong cur_pc = param->bb.cur_pc;
target_ulong cr3 = DECAF_getPGD(env);

if (DECAF_is_in_kernel(env) || !targetcr3s.count(cr3)) {
    current_cr3 = 0x00;
    return;
}
current_env = param->bb.env;
current_cr3 = cr3;
module = art_module = NULL;
art_module = VMI_find_module_by_pc(cur_pc, cr3, &base);

if (art_module != NULL &&
    (!strcmp(art_module->name, "system@framework@boot.oat") != NULL)) {
    if (!framework_offsets_extracted) {
        char* oat_file_str;
        extract_oat_file(env, base, &oat_file_str);
        extract_art_offsets_framework(base, art_module, env, cr3,
            std::string(oat_file_str));
        framework_offsets_extracted = true;
    }

    if (framework_offsets.count(((cur_pc - base - 0x1000)))) {
        printf(log_fd_jumps, "java_function_call=%s\n",
            framework_offsets[(cur_pc - base - 0x1000)].c_str());
        fflush(log_fd_jumps);
    }
}

if (art_module != NULL && (!strcmp(art_module->name, "libart") != NULL)) {
    if (funcmap_get_name_c(cur_pc, DECAF_getPGD(env), modname, functionname) == 0) {
        int reg_num = is_an_invoke_call(functionname);
        // Extract member offset from invoke.
        if (reg_num != -1) {
            // this pointer! artMethod
            target_ulong dex_cache, declaring_class,
            called_art_method = env->regs[0];

            // We need to dig one level deeper
            if (reg_num == 3) {
                target_ulong actual_art_method = 0x00;
                DECAF_end_mem_with_pgd(env, pgd_strip(cr3), called_art_method,
                    &actual_art_method, sizeof(target_ulong));
                called_art_method = actual_art_method;
            }

            // Get the ArtMethod
            art::mirror::ArtMethod* methoddz;
            char* block1 = sizeof_type(art::mirror::ArtMethod)];
    }
DECAF_read_mem_with_pgd(env, pgd_strip(c3), called_art_method, block1,
 SIZEOF_TYPE[art::mirror::ArtMethod]);
methodzz = {art::mirror::ArtMethod*}(block1);

// Get the ArtMethod's declaring class
art::MemberOffset declaring_class_offset =
methods->DeclaringClassOffset();
byte* raw_addr = reinterpret_cast<byte*>(methodzz) +
declaring_class_offset.Int32Value();
art::mirror::HeapReference<art::mirror::Class*> objref_addr =
reinterpret_cast<art::mirror::HeapReference<art::mirror::Class*>*>(
raw_addr);
declaring_class = {target_ulong)objref_addr->AsVRegValue();

art::mirror::Class* class = nullptr;
char block2 [SIZEOF_TYPE[art::mirror::Class]]; DECAF_read_mem_with_pgd(env, pgd_strip(c3), declaring_class, block2,
SIZEOF_TYPE[art::mirror::Class]);
class = {art::mirror::Class*}(block2);

// Get the Declaring class's DexCache
art::MemberOffset dex_cache_offset = class->DexCacheOffset();
raw_addr =
reinterpret_cast<byte*>(class) + dex_cache_offset.Int32Value();
art::mirror::HeapReference<art::mirror::DexCache*>
dexcache_objref_addr = reinterpret_cast<
art::mirror::HeapReference<art::mirror::DexCache*>*>(raw_addr);
dex_cache = {target_ulong}dexcache_objref_addr->AsVRegValue();

art::mirror::DexCache* dexcachezz = nullptr;
char block3 [SIZEOF_TYPE[art::mirror::DexCache]]; DECAF_read_mem_with_pgd(env, pgd_strip(c3), dex_cache, block3,
SIZEOF_TYPE[art::mirror::DexCache]);
dexcachezz = {art::mirror::DexCache*}(block3);

// Get the DexFile from the DexCache of the declaring class of the
// Artmethod
art::MemberOffset dex_file_offset = dexcachezz->GetDexFileOffset();
raw_addr =
reinterpret_cast<byte*>(dexcachezz) + dex_file_offset.Int32Value();
uint64_t* dex_file_ref = reinterpret_cast<uint64_t*>(raw_addr);

art::DexFile* dexfilezz = nullptr;
char block4 [SIZEOF_TYPE[art::DexFile]]; DECAF_read_mem_with_pgd(env, pgd_strip(c3), *dex_file_ref, block4,
SIZEOF_TYPE[art::DexFile]);
dexfilezz = {art::DexFile*}(block4);

// WE ARE DONE! WE GOT THE DexFILE! NOW TIME TO GET THE FUNCTION NAME +/

// Try to grab all methods from the dex files!
// This process is similar to what is done in the DexMethodIterator
raw_addr = reinterpret_cast<byte*>(dexfilezz) + 4;
```c
uint32_t* begin_decaf = reinterpret_cast<uint32_t*>(raw_addr);
target_ulong dex_begin = (target_ulong)(uintptr_t)*begin_decaf;

module* dirty_module = VMI_find_module_by_pc(
    reinterpret_cast<target_ulong>*begin_decaf, cr3, &base);

if (dirty_module != NULL &&
    strstr(dirty_module->name, "framework") == NULL) {
    extract_art_offsets__(base, dirty_module->size,
        std::string{dirty_module->name}, env, cr3);
} else if (dirty_module == NULL) {
    target_ulong prev_end = 0x00;
    // monitor_printf(default_mon, "unknown module \n");
    // reinterpret_cast<target_ulong>*begin_decaf);
    dirty_module =
        VMI_find_next_module(reinterpret_cast<target_ulong>*begin_decaf),
        cr3, &base, &prev_end);
    extract_art_offsets__(_prev_end, base - prev_end,
        std::string{dirty_module->name}, env, cr3);
}

if (dirty_module && strstr(dirty_module->name, "framework") != NULL)
    return;

if (reg_num == 3)
    return;

/* Here we try to replicate the process used in
   DexFile::GetMethodName(MethodId)
   The process goes something like this
   -> From MethodId get the offset of the name of method in the
   StringIds
   -> Extract the exact StringId from this offset
   -> Use this StringId to find the offset of the actual string
   in the DexFile from the base of the desfile
   */

raw_addr = reinterpret_cast<BYTE*>(dexfilesz) + 8;
uint32_t* dex_file_size = reinterpret_cast<uint32_t*>(raw_addr);

// Used to extract code item
art::MemberOffset dex_code_item_offset =
    methods->GetCodeItemOffset();
raw_addr = reinterpret_cast<BYTE*>(methods) +
    dex_code_item_offset.int32Value();
uint32_t* code_item_offset = reinterpret_cast<uint32_t*>(raw_addr);

// This is the offset of the method in the MethodIds array
art::MemberOffset dex_method_id_offset =
    methods->GetMethodIndexOffset();
raw_addr = reinterpret_cast<BYTE*>(methods) +
    dex_method_id_offset.int32Value();
uint32_t* dex_method_id = reinterpret_cast<uint32_t*>(raw_addr);

// This is to get the base of the MethodIds array and add the offset to
```
// get the appropriate MethodId member
art::MemberOffset dexFile::method_ids_offset =
  dexFilezz->GetMethodIdsOffset();
raw_addr = reinterpret_cast<.byte*>(dexFilezz) + 48;
uint32_t* ids_decaf = reinterpret_cast<.uint32_t*>(raw_addr);
art:: DexFile::MethodId* temp_id = (art:: DexFile::MethodId*)ids_decaf;
  temp_id = temp_id + &dex_method_id;
art:: DexFile::MethodId* idzz;
char block5[SIZEOF_TYPE[art:: DexFile::MethodId]];
DECAF_rmd_mem_with_pgd(env, pgd_strip(cr3),
  { target_ulong }[uintptr_t]temp_id, block5,
SIZEOF_TYPE[art:: DexFile::MethodId]);
idzz = (art:: DexFile::MethodId*)block5;

// Now we have the MethodId in 'idzz', and idzz->name_idx holds the
// offset of the StringId
// Proceed getting the StringId
raw_addr = reinterpret_cast<.byte*>(dexFilezz) + 48 - 12;
uint32_t* str_ids_decaf = reinterpret_cast<.uint32_t*>(raw_addr);
art:: DexFile::StringId* temp_str_id =
  (art:: DexFile::StringId*)str_ids_decaf;
  temp_str_id = temp_str_id + idzz->name_idx;
art:: DexFile::StringId* str_idxz;
char block6[SIZEOF_TYPE[art:: DexFile::StringId]];
DECAF_rmd_mem_with_pgd(env, pgd_strip(cr3),
  { target_ulong }[uintptr_t]temp_str_id, block6,
SIZEOF_TYPE[art:: DexFile::StringId]);
str_idxz = (art:: DexFile::StringId*)block6;
  // We now have the StringId at str_idxz, PHEW!!
char block7[200];
DECAF_rmd_mem_with_pgd(env, pgd_strip(cr3),
  dex_begin + str_idxz->string_data_off_ + 1,
  block7, 200);
block7[199] = '\0';

if (reg_num == 0) {
  fprintf(log_fd_jumps,
    "java_function_call_=\%s\n",
    block7);
  fflush(log_fd_jumps);
  return;
} else {
  fprintf(log_fd_jumps, "java_function_call_=\%s\n",
    reinterpret_cast<.target_ulong*>(begin_decaf));
art:: DexFile::CodeItem* this_code_item;
char code_item_block[SIZEOF_TYPE[art:: DexFile::CodeItem]];
DECAF_rmd_mem_with_pgd(env, pgd_strip(cr3),
  dex_begin + &target_ulong)(*code_item_offset),
  code_item_block,
SIZEOF_TYPE[art:: DexFile::CodeItem]);
this_code_item = (art:: DexFile::CodeItem*)code_item_block;
uint32_t num_bytes_to_read =
    this_code_item->insns_size_in_code_units + 2;

target_ulong to_check_start =
    dex_begin + (target_ulong)(+code_item_offset);
target_ulong to_check_end =
    dex_begin + (target_ulong)(+code_item_offset) +
    SIZEOF_TYPE(DexFile::CodeItem) + num_bytes_to_read + 4;

while (to_check_end != to_check_start) {
    if (byte_addrs_written.count(to_check_start)) {
        fprintf(log_fd_jumps, "<dirty_dalvik_code>
"
        break;
    } ++to_check_start;
    }
    fflush(log_fd_jumps);
    }
    }
    }
    }

end:
if (base_to_offsets.count(base)) {
    std::unordered_map<target_ulong, std::string>& oat_module_offsets =
        base_to_offsets[base];

    std::unordered_map<target_ulong, target_ulong>& oat_module_sizes =
        base_to_sizes[base];

    if (oat_module_offsets.count([cur_pc = base - 0x1000])) {
        increment_something("num_native_methods");
        fprintf(log_fd_jumps, "java_function_call_%s\n",
            oat_module_offsets[([cur_pc = base - 0x1000]).c_str()]);

    target_ulong native_method_size =
        oat_module_sizes[([cur_pc = base - 0x1000])];
    target_ulong native_method_end = cur_pc + native_method_size,
        native_method_begin = cur_pc;

    while (native_method_end != native_method_begin) {
        if (byte_addrs_written.count(native_method_begin)) {
            fprintf(log_fd_jumps, "<dirty_native_code>\n"
            break;
        } ++native_method_begin;
        }
    fflush(log_fd_jumps);
    }
    }
    }
}
static void createproc_callback(VMI_Callback_Params* params) {
if (targetpid == 0 && strlen(targetname) > 1 &&
    strstr(params->cp.name, targetname) != 0) {
    targetpid = params->cp.pid;
    targetcr3 = params->cp.cr3;
    targetcr3s.insert(targetcr3);

    strncpy(actualname, params->cp.name, strlen(params->cp.name));
    actualname[511] = '0';

    register_hooks();
    DECAF_printf("process found : pid=%08x, cr3=%08x, name=%s\n", targetpid,
                 targetcr3, params->cp.name);
}
else if (targetpid != 0 && params->cp.parent_pid == targetpid) {
    targetcr3s.insert(params->cp.cr3);
    DECAF_printf("child process found : pid=%08x, cr3=%08x, name=%s\n",
                 params->cp.pid, params->cp.cr3, params->cp.name);
    increment_something("child processes");
}
}
Appendix B

Results.

The raw results for the experiments can be obtained from

https://gitlab.com/TheLoneRanger14/thesis_results.git

The password to the zip is ‘droidunpack‘ and the zip contains a README describing the structure of the results.
Vita

Abhishek Vasisht Bhaskar, was born in Bangalore, India to parents Girija Bhaskar and H.N.Bhaskar. After finishing his undergraduate program in Telecommunication engineering at PES University in Bangalore he arrived at Syracuse, NY to pursue graduate studies. As of July 2016 he graduates from a Masters program in Computer engineering from the EECS department at Syracuse University. During his time at Syracuse University, he served as a Research Assistant under Dr. Heng Yin as a part of the Systems Security Lab at Syracuse University for little over a year. This work was thought, implemented and verified during the same period.

Current Location: Syracuse, NY 13210.

This thesis was typed by the author.
Bibliography


