A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART)

Alberto Magno Muniz Soares and Rafael Timóteo de Sousa Jr.
Eletrical Engineering Department, University of Brasília (UnB), Brasília, Brazil
alberto.magno@gmail.com, desousa@unb.br

Keywords: Mobile Device Forensics, Memory Forensics, Memory Analysis, Android.

Abstract: This paper describes a technique for analysing objects in memory within the execution environment Android Runtime (ART) using a volatile memory data extraction. A study of the AOSP (Android Open Source Project) source code was necessary to understand the runtime environment used in the modern Android operating system, and software tools were developed allowing the location, extraction and interpretation of useful data for the forensic context. Built by the authors as extensions for the Volatility Framework, these tools help to locate, in a memory extraction from a device compliant with the ARM architecture, arbitrary instances of classes and their data properties.

1 INTRODUCTION

Personal mobile devices can be used for many purposes and so its RAM may contain digital evidence for a potential investigation.

Traditionally, forensics on mobile devices focus on the acquisition and analysis of data present in non-volatile storage media. Usually, depending on the purpose of the investigation or given the difficulty with the ephemeral nature of the data, volatile memory exams are not performed. On the other hand, with the increasing use of encryption and the presence of ever more sophisticated malicious software, the need to conduct investigations on the volatile memory contents of mobile devices has become even more important.

Also, as discussed in (Brezinski and Killalea, 2002), the forensic community seems to recognize that capturing data in memory is required in order to comply with the volatility of digital evidence, since some information about the system environment are never kept statically in secondary storage media. Thus, it has become imperative to use techniques to analyse data from a volatile memory extraction, going further than traditional techniques.

Android is an operating system based on the Linux kernel and is designed especially for mobile devices. Currently this system leads the mobile operating systems market, with versions for 32-bit and 64-bit processors, complying with x86, MIPS and, especially, the ARMS architecture. Despite being a Linux distribution, it has features that require a detailed understanding of the runtime environment and the use of specific techniques for extraction and memory analysis.

As published in the Android Open Source Project (AOSP), Android OS in version 5.0 contains a new runtime environment (Android Runtime - ART) operating in most available devices, replacing the interpretation mechanism of the former Dalvik Virtual Machine (DVM). In place of an interpretation engine ART requires compilation of every application during installation, a process that is called Ahead-Of-Time (AOT). Also, this new runtime environment comes with new memory management mechanisms.

The general digital forensics process includes the acquisition of data from a source, the analysis of the data and extraction of evidence, with the preservation and presentation of the evidence (Carrier, 2003). In spite of several RAM memory data acquisition techniques exist for Android, a forensic technique specific for memory analysis and extraction of Java objects in the ART runtime environment is yet to be established. Thus, the central contribution of this paper is to address a technique for memory analysis based on the source code available from AOSP. This proposed and tested technique allows the location and extraction of object data of a running application, using the content of volatile memory acquired in Android 5.0

Soares, A. and Jr., R.
A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART).
DOI: 10.5220/0006204101470156
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 147-156
ISBN: 978-989-758-209-7
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

devices. Another contribution is the development of software tools that support the proposed forensic technique.

The remaining of this paper is organized as follows. Section 2 describes related work. Section 3 is an overview of the Android architecture, while Section 4 is devoted to ART. In Section 5, the proposed forensic technique is described with its supporting tools. Section 6 discusses results from the experimental evaluation of the proposed technique and developed support tools, for RAM acquisitions from an emulated device and in real one, and Section 7 presents conclusions and possible future works.

2 RELATED WORK

As discussed in (Wächter and Gruhn, 2015), the feasibility of acquisition techniques for forensic purposes has limitations related to intrinsic features implemented by manufacturers, such as hard security mechanisms that prevent access to data.

Nevertheless, there are different known techniques for RAM acquisition in Android, a well-known one called Linux Memory Extractor - LiME (Sylve et al., 2012), which extracts raw data from volatile memory of a device ensuring a high integrity rate in its results.

In (Apostolopoulos et al., 2013), a study is presented on recovery of credentials from Android applications by means of available volatile memory extraction techniques. This study shows that even without the analysis of applications objects, the referred credentials are accessible by direct inspection of the extracted data. But the analysis of data extracted from real devices and from emulated systems showed no large discrepancies in the results.

As an alternative to bypass hard security barriers, a work is presented in (Hilgers et al., 2014) based on data extraction of real devices with Android version below 4.4, using the technique called Forensic Recovery of Scrambled Telephones – FROST. This paper holds that, even in case of rebooting and unrecoverable data erasure in non-volatile memory, which occurs in some devices when they are reset to factory state, a situation caused by the bootloader unlocking process, it is still possible to analyse the remaining data in RAM, including Java objects maintained by the old Dalvik runtime, a process that is made using plugins of the Volatility Framework (http://www.volatilityfoundation.org).

In (Backes et al., 2016) the compilation process and instrumentation solutions for applications within ART are presented, highlighting innovations in the ART compilation process, including significant internal operation details that are useful in understanding the difference between the ART and the earlier Android runtime versions.

After careful publications search, we verified that forensics studies on ART for Android version 5.0 or greater are still rare. Then, the analysis of the AOSP code and its constant updates is an important source of information.

3 ANDROID ARCHITECTURE OVERVIEW

The Android platform consists of a software stack with three main layers: an application layer, one layer containing a framework of Java objects and the Runtime environment - RT, and a native code Linux kernel layer containing hardware abstraction libraries (Yaghmour, 2013).

Regarding the memory management used by the RT, as described in (Drake et al., 2014), the Android system does not offer a memory swap area, but instead it uses paging mechanisms and file mapping.

Regarding the paging mechanism, page sharing is used between processes. Each process is instantiated by fork of a pre-existing process called Zygote. This process starts during the system initialization phase (boot) and loads the code and features that are part of the Android framework. This allows many pages, allocated to the code and resources of the framework, to be shared by all other process applications.

With the mapping mechanism, most of the static data (byte-code, resources and possible native code libraries) of an application are mapped into the memory address space of the application process. This allows data sharing between processes and the concerned memory pages can be disposed as needed. Memory sharing between applications works through an asynchronous sharing mechanism called Anonymous Shared Memory (Ashmem). Ashmem is an additional modification of the Android Linux kernel to allow automatic adjustment of the size of memory caches and recover areas when the total available memory is low (Yaghmour, 2013). Also, by means of a memory snapshot, the virtual memory area of an application may present the unused mapped pages.

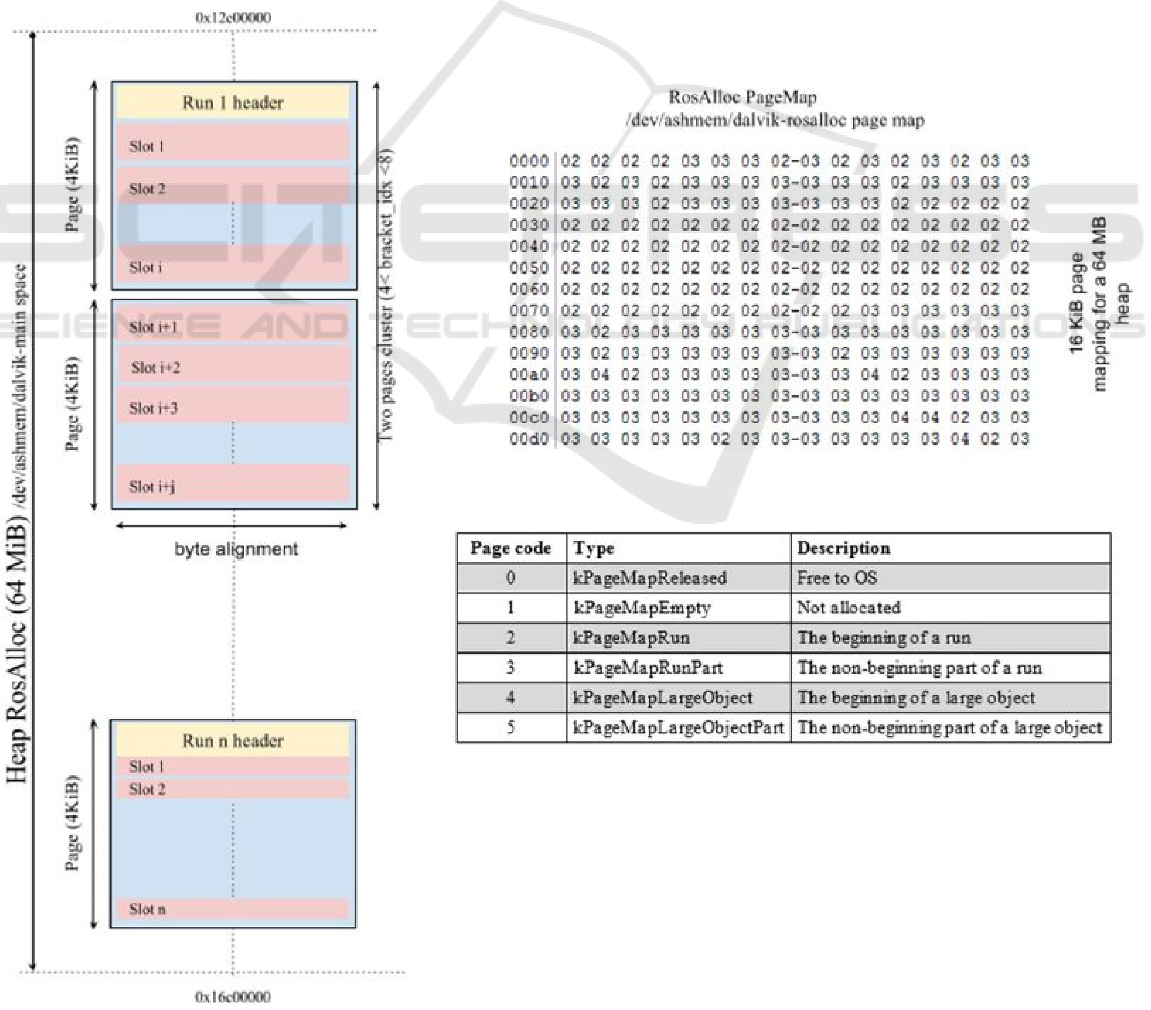
In the boot process, in addition to the preparation of the Zygote by the RT process, a service starts keeping (for each boot) memory mapping pairs of key-value related to system configuration, comprising data properties files and other sources of the operational system. Many components of the operating system and the Android framework, including the execution environment, use these values, including those related to the configuration of the execution environment (for instance, the size of the memory space for the Java object heap and parameters of the Garbage Collection - GC).

With respect to security in AOSP, after installation, each application is activated in its own virtual memory area, implementing the principle of least privilege. Android version 5.0 includes security mechanisms that require that all dynamic code liking being of relative type (Position-Independent Code - PIC), reinforcing the existent mechanism of Address Space Layout Randomization (ASLR).

Despite operating on a Linux kernel, these peculiar characteristics of the Android architecture with respect to the security mechanisms, memory management, and application runtime environment, impose the use of specific techniques in the RAM extraction and analysis procedures.

4 ANDROID RUNTIME (ART)

The runtime module is responsible for managing Android applications designed to operate on the Android framework layer. One of its responsibilities is to provide memory management for application execution and access to other system services such as Virtual Machine (VM) byte-code compilation and loading (in DEX files). This VM is similar to a Java Virtual Machine (JVM) and runs as an application that in ART keeps the name and uses the same byte-code of Dalvik, despite of the replacement of the corresponding legacy runtime module.



RosAlloc PageMap
/dev/ashmem/dalvik-rosalloc page map

```
0000 02 02 02 02 03 03 03 02-03 02 03 02 03 02 03 03
0010 03 02 03 02 03 03 03 03-03 03 03 02 03 03 03 03
0020 03 03 03 02 03 03 03 03-03 03 03 02 02 02 02 02
0030 02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
0040 02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
0050 02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
0060 02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
0070 02 02 02 02 02 02 02 02-02 02 03 03 03 03 03 03
0080 03 02 03 03 03 03 03 03-03 03 03 03 03 03 03 03
0090 03 02 03 03 03 03 03 03-03 02 03 03 03 03 03 03
00a0 03 04 02 03 03 03 03 03-03 03 04 02 03 03 03 03
00b0 03 03 03 03 03 03 03 03-03 03 03 03 03 03 03 03
00c0 03 03 03 03 03 03 03 03-03 03 03 04 04 02 03 03
00d0 03 03 03 03 03 02 03 03-03 03 03 03 03 04 02 03
```

Page code	Type	Description
0	kPageMapReleased	Free to OS
1	kPageMapEmpty	Not allocated
2	kPageMapRun	The beginning of a run
3	kPageMapRunPart	The non-beginning part of a run
4	kPageMapLargeObject	The beginning of a large object
5	kPageMapLargeObjectPart	The non-beginning part of a large object

Figure 1: Example of a heap structure and mapping maintained by RosAlloc.

Previously to running applications, the ART initializes a set of classes during the first boot (or after system modifications), generating a file that contains an executable image with extension "art " with all the loaded classes and objects that are part of the Android framework. This file, called boot.art, is mapped into memory during the Zygote boot process, and basically contains natively compiled objects who hold address references (pointers) with absolute addresses within the image itself and references to methods in the code contained in framework files (inside the framework file there are absolute pointers to the image as well). The overall data structure related to the compilation and execution in the ART environment is then described in the image header, including a field that stores the respective offset from the beginning of the file. This value changes at every boot so that the image is not loaded at the same address (in AOSP version 5.0, the base address for the displacement of ASLR was set to 0x70000000).

After the initial preparation, the byte-code of each installed application is compiled to native code before its first run. The product of this compilation, comprising each application byte-code and libraries that make up the Android framework, are files in Executable and Linking Format - ELF, called OAT (specifically boot.oat for the framework). These files, compiled to boot the Android framework and to install applications, contain three dynamic symbol tables called oatdata, oatexec and oatlastword that respectively contain the OAT header and DEX files, the compiled native code for each method, and the last 4 bytes of generated native code functioning as a final section marker.

For memory management, the ART divides the virtual memory as follows: a main space for application’s Java objects (heap), a space for the image objects and classes of the Android framework, a space for Zygote’s shared objects, and a space for large object (Large Objects Space – LOS). The first three are arranged in a continuous address space while there is a collection of discontinuous addresses for the LOS. In addition to these spaces, there are data structures related to garbage collection whose types are related to the GC and the Java object heap allocation and that can be active depending on the GC plan that is working. The GC plan is usually set by the manufacturer according to the device's intrinsic characteristics and according to the plan established by the memory allocator. For devices such as common use smartphones, without strong memory constraints, there is generally a defined plan whose operating mode works with the allocator called Runs-Of-Slots-Allocator (RosAlloc) for mutable objects and with Dlmalloc for immutable objects.

The RosAlloc came up with the ART runtime environment, and is the main allocator responsible of heap memory space for Java objects. It organizes this memory space in rows of slots of the same size. These runs are clustered as pages within brackets. The first page of a bracket contains a header that determines the number of pages this bracket contains and the slot’s allocation bitmap. The number of slots per page is set according to the size of the bracket, the header length and the byte alignment (which depends on the target device architecture). Figure 1 illustrates an example of a heap structure and mapping schema. Each slot stores data for one object and the first bytes store its parent class address. The slot is classified according to the size of the object as a means to reduce fragmentation and allow parallel GC. Objects with big data (≥ 12 KiB) are spread through LOS allocation areas, allowing the kernel to conveniently manage address spaces to store this data.

The allocator maintains an allocation map for the brackets pages (each page with 4 KiB size) setting in this map the type of each page in the allocation space. This map is stored in a mapped file in RAM (rosalloc page map). For the allocation of the heap space, it sets the address to start near the lowest virtual address of the process, from 0x12c00000 bytes (300 MiB).

Considering this memory layout information, drawn from our analysis of the AOSP source code, it is possible to establish a strategy for locating objects by scanning the bracket's slots inside the heap mapped file and decoding the data set for each allocated object. This is also possible for a recoverable object from a deallocated slot. While these are subjects of the present paper, as approached in the next section, the analysis of data stored in structures related to large objects or allocated by native libraries, which have specific allocation mechanisms, are considered for future work.

5 OBJECT ANALYSIS TECHNIQUE

As exposed above, in an application’s runtime environment there are mapped files in RAM containing: information about system properties, Android framework, Java object heap, mapping of objects used by the memory allocator, as well as class definitions and executables compiled from the application's DEX files contained by OAT files.

From a whole RAM extraction, the technique proposed in this paper, as illustrated in Figure 2, is aimed at recovering Java objects for data analysis from the heap space. This is performed by inspecting the mapping maintained by the memory allocator, based on the premise that from a volatile memory extraction it is possible to recover data pages from those files. For Java objects data, according to the type of the page (guided by the mapping maintained by the allocator file) and the respective page header data, it is possible to recover the slots and, with the appropriate description of the target object class, decode the data.

Object data decoding can be performed directly or from the traversal of the references throughout the class hierarchy (similar to a recursive programming process) using memory layout information obtained by decompilation of the application byte-code or by understanding the upper classes information. In Figure 3, a generic sequential process for recovering an arbitrary string field of the Object X is illustrated. From Object X slot (bottom-left in figure), it is possible to walk through the parent classes references, this way decoding object data using the layout of known Android framework classes.

The Volatility Framework (in version 2.4), described in (Ligh et al., 2014), provides tools and data structures mappings with support for the Linux platform on the ARM 32-bit architecture, allowing the retrieval of information, such as process table and memory mapping, among others. In this paper, the process of data analysis is supported by a set of tools that were conjointly developed within the Volatility Framework, based on Android AOSP source code for ART version 5.0.1_r1 (https://android.googlesource.com/platform/art/+/android-5.0.1_r1), and on ART related information described in (Sabanal, 2014/2015).

These extensions built for the Volatility framework allow retrieval of information on the execution environment and the recovery of allocated Java objects. For the recovery of the runtime data structures, we have created mappings for interpretation of data from ART files, OAT, DEX, Java framework classes, heap pages structures and system properties. Then, for the extraction and analysis process, we have built tools for recovery of the runtime properties, location of OAT files, data decoding from DEX files, extraction of Java objects from the heap, and for decoding object data from the heap and from the Android framework image. The architecture of the Volatility framework and the design of these tools allow updating and adding new mappings, which facilitate adaptation to other architectures or changes in future versions of Android.

SCIENCE AND TECHNOLOGY PUBLICATIONS

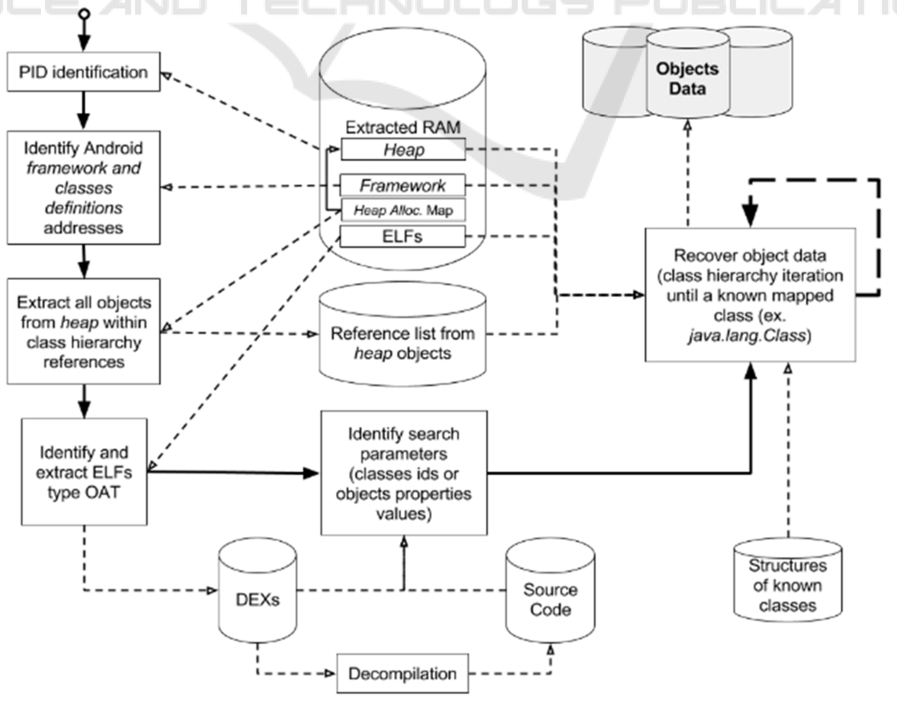

PID identification
Identify Android framework and classes definitions addresses
Extract all objects from heap within class hierarchy references
Identify and extract ELFs type OAT
Extracted RAM
Heap
Framework
Heap Alloc. Map
ELFs
Reference list from heap objects
Identify search parameters (classes ids or objects properties values)
Objects Data
Recover object data (class hierarchy iteration until a known mapped class (ex. java.lang.Class)
DEXs
Source Code
Decompilation
Structures of known classes

Figure 2: Analysis technique for heap objects maintained by RosAlloc.

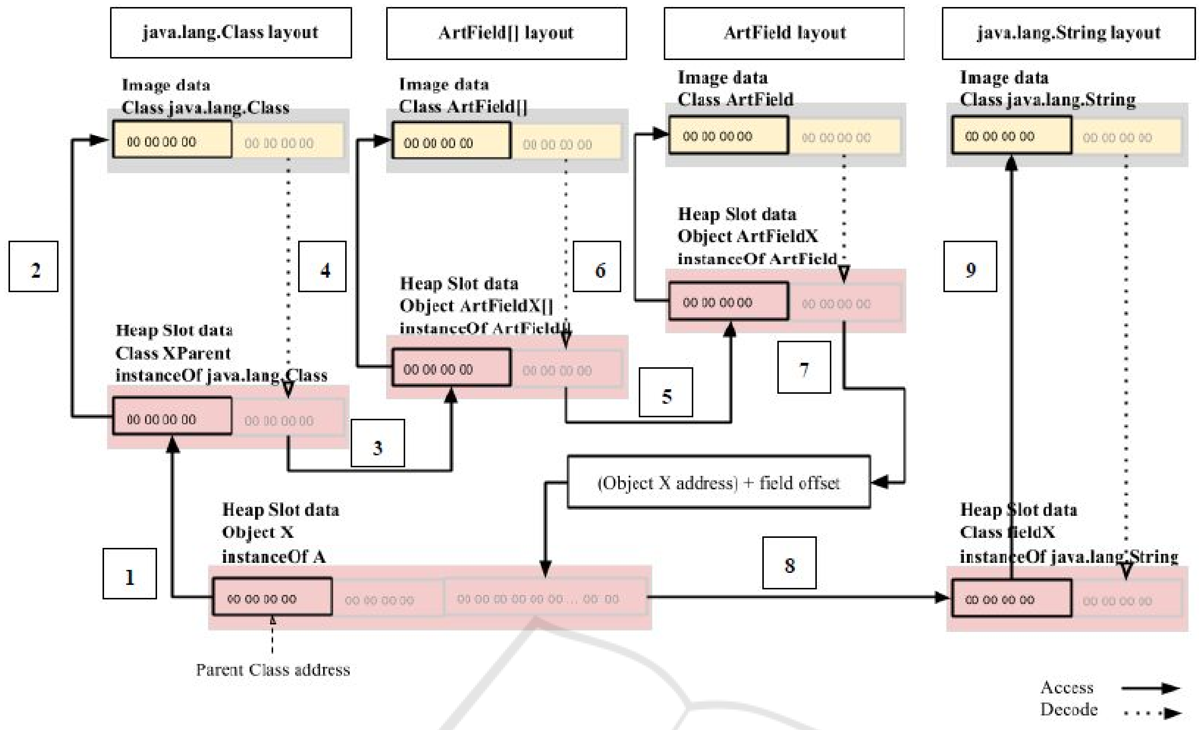


Figure 3: Recovery example of an object field.

The list of references to heap objects used in data extraction is constructed by inspecting and decoding the slots of the heap pages described in the mapping file maintained by the RosAlloc allocator. This list contains data objects with the location of the object (address, page, bracket, and slot), the parent class, class identifiers in DEX, and raw or textual data (of type String or char array).

This technique enables in-depth analysis of the extracted data, overcoming the traditional techniques of carving, text or other articles search, which lack the understanding of the storage structures in memory.

6 EXPERIMENTAL EVALUATION

The experimental evaluation of the proposed technique was done for an emulated device and for a real one, both representative of a common ART environment. A complete RAM memory dump from each device was acquired using the technique described in (Sylve et al., 2012), while these devices were running with active applications, including a chat application (WhatsApp v.2.12.510)

For acquiring memory dumps from both devices, it was necessary to use a privileged user access (root) and to perform the replacement of the kernel code with a newly built compilation configured to accept loading kernel modules without validation.

The privileged user is available by default in the emulator, while for the real device it was obtained using the rooting tool Kingo (http://www.kingoapp.com).

The source code of the kernels was compiled according to the guidelines in the AOSP site. The workstation used for the process of cross-compilation and analysis consists of the Santoku Linux version 0.4, with the installation of the Android NDK (Release 8e) and the Volatility Framework (version 2.4), as described in their project sites. The configuration was based on the construction of the experimental setting procedure used in (Høgset, 2015). For each memory acquisition, the RAM memory data was transferred by TCP directly to the analysis workstation.

6.1 Evaluation with an Emulated Device

The emulated device is an Android Virtual Device (AVD) configured with the parameters CPU / ABI: ARM (armeabi-v7a), 768 MB RAM, Target: Android 5.0.1 (API level 21), build number “sdk_phone_armv7-eng 5.0.2 LSY64 1772600 testkeys”, hw.device.name Nexus 5 and vm.heapSize 64 MB.

6.1.1 Environment Set-up

The target device used for memory acquisition is the Android emulator, available in the development tools package Android SDK Tools Revision 23.0.2.

The source code of the kernel version (3.4.67) available for the emulator (goldfish) was obtained from the AOSP site.

6.1.2 Evaluation

In analyzes of the memory extraction according to the proposed technique, it is possible to successfully recover common interesting forensic data from ART objects, as for instance the user contacts maintained by the *com.android.contacts* application.

For a deeper evaluation example, we describe hereafter in detail how to discover and characterize the objects from a running chat application (*com.whatsapp* v. 2.12.510) involving messages exchanged with another user in a real device.

Initially, the extension to the Linux Volatility framework that allows retrieving the table of running processes is used. Thus, it is possible to locate the target process for the analysis which in this case is identified by PID 1206. With the developed tool for recovery of system properties, environmental data is extracted, including the size of the heap for Java objects, a value that subsequently is used as a parameter in the recovery of target objects:

```
>python vol.py --
profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime
art_extract_properties_data -p 1206

...
[dalvik.vm.heapsize]= [64m]
...
```

Then, it is possible to retrieve data about the application execution environment, such as the addresses related with the Android framework mapping, using the tool built for this activity and the target process handle as a parameter:

```
>python                vol.py                 --
profile=LinuxLinuxGoldfish_3_4_67ARM          -f
memdumpWhatsAppChat.lime  art_extract_image_data
-p 1206

com.whatsapp
ART image Header
-----------------------
image_begin:0x700c7000
oat_checksum:0xbd5a21c9L
oat_file_begin:0x70be8000
oat_data_begin:0x70be9000
…
image_roots:0x70bb8840
…
   kClassRoots:0x70bb8948
       0x1 LJava/lang/Class; 0x700c7220L
       0x2 LJava/lang/Object; 0x700f7240L
…
       0x5 LJava/lang/String; 0x700df8f0L
       0x6 LJava/lang/DexCache; 0x700c74f0L
…
       0x8 LJava/lang/reflect/ArtField;
0x700f7640L
…
…      0xc [LJava/lang/reflect/ArtField;
0x700f74a0L
       0x1d [C 0x700f6fd8L
…
```

The recovered information present in the image header, including the memory offset for the location of the mapping framework (0x700c7000), serves as the basis for recovering addresses from various classes, such as *java.lang.String* class. With these data and the map maintained by the RosAlloc allocator, the list of heap objects containing references to object data and references to other objects is constructed, also using a developed tool. The address allocation map (0xb1d70000) is recovered by searching the name of the respective file in the mapping process.

With this gathered information, and by means of another developed localization tool, it is possible to recover OAT files used by the target process, including the addresses of each location in the addressing process:

```
>python                vol.py                 --
profile=LinuxLinuxGoldfish_3_4_67ARM          -f
memdumpWhatsAppChat.lime art_find_oat -p 1206

Oat                                      offset_
--------------------------------------- ----------
webview@webview.apk@classes.dex          0xa06dc000L
com.whatsapp-1@base.apk@classes.dex      0xa5a74000L
```

With the OAT address, it is possible to recover data that enables a static analysis of some components, including class identifier indexes and application's byte-code. After analyzing the OAT decompiled code of the file located in 0xa5a74000L, comes the selection of the identifier (DEX_CLASSDEF_IDX = 0x1394) for the class of objects (*com.whatsapp.protocol.l*) that indicates the storage for the target application messages text data.

Searching the list of heap objects references, looking for references to the definition of the requested class, it is possible to identify the parent class *java.lang.Class* object (described in the Android framework image at 0x700c7220L):

```
>python                vol.py                   --
profile=LinuxLinuxGoldfish_3_4_67ARM            -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000
art_dump_rosalloc_heap_objects -e 0x12c00000 -m
0xb1d70000 -s 0x4000000

address       page bracket slot obj class
------------- ---- ------- ---- ----------------
0x1384d2c0L   3149  13      2 *(FOUND)* 0x12c19020
0x1384e0c0L   3149  13     18 *(FOUND)* 0x12c19020
0x1384f240L   3149  13     38 *(FOUND)* 0x12c19020
0x13850820L   3149  13     63 *(FOUND)* 0x12c19020
```

Then, using a developed tool for object data recovery, it is possible to examine the data for each specific object of this class, i.e., data recovery is made for the object located in 0x1384d2c0:

```
>python vol.py --
profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime  -p 1206 -b 0x700c7000
art_extract_object_data -o 0x1384d2c0

Object Address: 0x1384d2c0
Class Address: 0x12C19020
…
Loaded: 0x700c7220L
LJava/lang/Class;
classLoader 0x12c02b20L
componentType 0x0L
dexCache 0x12c01610L LJava/lang/DexCache;

directMethods 0x133ff980L
[LJava/lang/reflect/ArtMethod;

iFields 0x12c04900L
[LJava/lang/reflect/ArtField;
..
sFields 0x13407500L
[LJava/lang/reflect/ArtField;

dexClassDefIndex 0x1394L
dexTypeIndex 0x1810L
…
```

Among the recovered data, the address with reference to the array of properties *java.lang.reflect.ArtField[]* (at 0x12c04900L) is found. With a new search to this address and for this type of class, data from the conversation, including the message text, is recovered. By tracking through references and properties of the recovered objects of this class other attributes are identified: text, date, peer ID, and other data.

Figure 4 illustrates the links between some of the addresses visited for retrieval of data objects related to the target object. It is noteworthy that the developed tools also support the reverse process which, given a specific object property (e.g. message text), reveals references of objects related to the concerned chat.

6.2 Evaluation with a Real Device

The real device specification was a Samsung Galaxy S4 (GT-I9500 non-LTE) with CPU Exynos 5410, 2 GB RAM, original Android 5.0.1 (API level 21), build number LRX22C.I9500UBUHOL1, and vm.heapSize 64 MB.

6.2.1 Environment Set-up

The cross-compiled kernel source code (version 3.4.5) was obtained from the manufacturer open source release site (http://opensource.samsung.com).

6.2.2 Evaluation

Initially, the procedure to locate the target process (*com.whatsapp*) is executed and retrieves data about the application execution environment, such as the addresses related to the Android framework mapping.

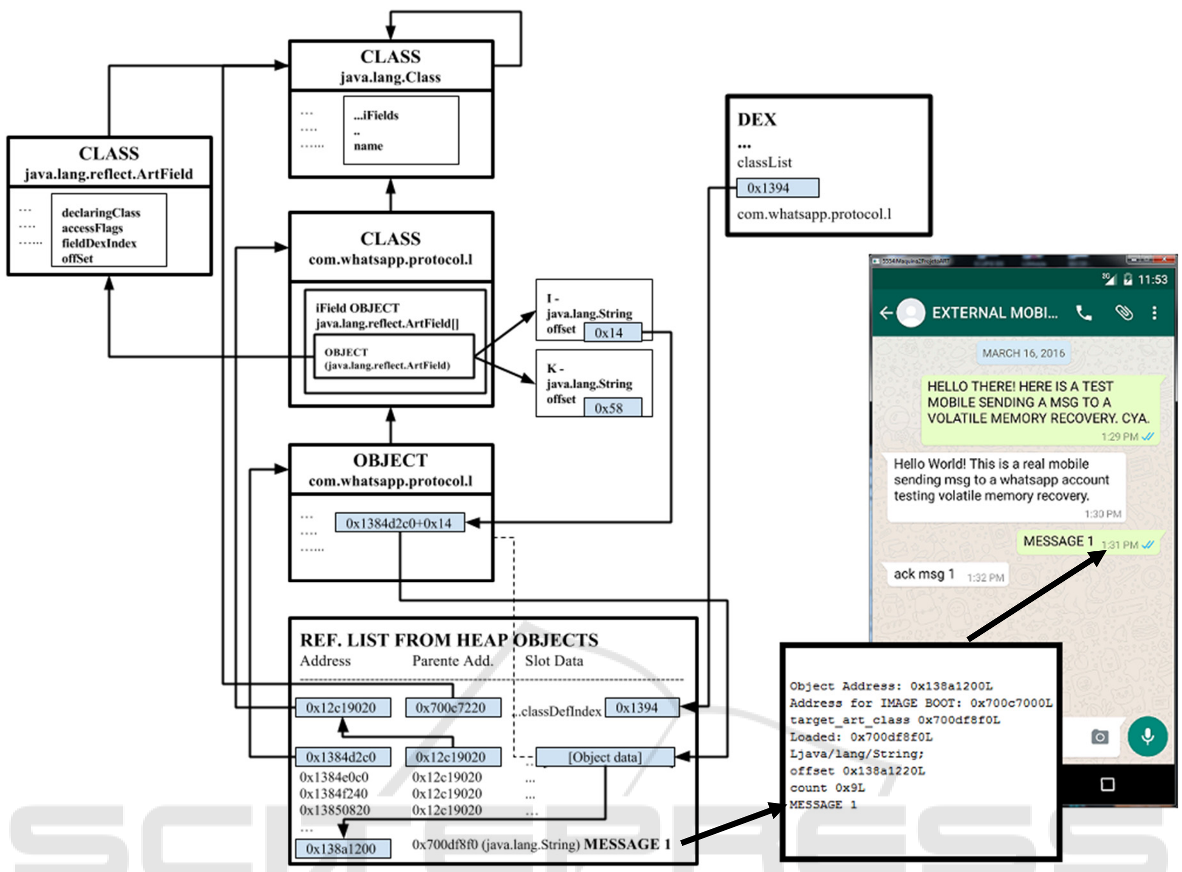
Then, it is interesting to find that the Android framework image header in this device is different from that in the emulated device, although this real system presents the same ART header version identification (009).

In the real device, the header field for the image address does not point to a valid absolute address in the image segment. This difference suggests that this manufacturer Android OS does not correspond to the AOSP source code.

Consequently, the technique proposed in this paper cannot be fully used in this case since the unknown header demands reverse engineering the ART image present in this real device. This evaluation result shows a common limitation characterizing procedures designed for extraction of objects from ever evolving operating systems in mobile devices. Moreover, the consequent requirement regarding the adaptation of the proposed technique to this new situation comes up against an important obstacle, since there is no available public ART runtime source code provided by the concerned manufacturer.

7 CONCLUSIONS AND FUTURE WORK

This paper presents a technique for object data analysis in RAM acquisitions from devices compliant to the ARM 32-bit architecture. The work includes the study of concepts and structures of the ART runtime environment, present in the Android



HELLO THERE! HERE IS A TEST MOBILE SENDING A MSG TO A VOLATILE MEMORY RECOVERY. CYA. 1:29 PM

Hello World! This is a real mobile sending msg to a whatsapp account testing volatile memory recovery. 1:30 PM

MESSAGE 1 1:31 PM

ack msg 1 1:32 PM

```
Object Address: 0x138a1200L
Address for IMAGE BOOT: 0x700c7000L
target_art_class 0x700df8f0L
Loaded: 0x700df8f0L
Ljava/lang/String;
offset 0x138a1220L
count 0x9L
MESSAGE 1
```

Figura 4: Example of references to objects related with a message text recovery.

operating system version 5.0 from AOSP. Experimental evaluation of the proposed technique is performed using software extensions developed for the Volatility framework.

The proposed technique contribution comes from its ability to extract and analyse Java objects in ART revealing involved memory structures, thus overcoming earlier Dalvik analysis (Hilgers et al., 2014) and other traditional techniques based on detecting patterns intrinsic to the artefact components. An additional contribution concerns the supporting tools developed as Volatility plugins that can also be useful as reverse-engineering tools being soon available for the forensic community.

It is noteworthy that the proposed technique and the constructed support tools have the flexibility to be adapted to other computer architectures (including 64-bit), for devices with different hardware limitations and to comply with ART modifications already identified in the AOSP source code of the latest versions of Android (6.0). It is relevant that, though it is successful in analysing heap objects from ART in an emulated device, the technique identifies an implementation of ART in a real device that differs from the AOSP version tested in an emulated device.

As future work, the authors intend to carry out the experimental validation of the technique with data retrieved from other real devices, and to associate the technique with similar ones for detection and analysis of malwares.

ACKNOWLEDGEMENTS

Authors wish to thanks the following Brazilian Research, Development and Innovation Agencies CAPES (project FORTE 23038.007604/2014-69) and FINEP (project RENASIC/PROTO 01.12.0555.00), as well DITEC/DPF/MJ (Contract 36/10 DPF-FUB) and IC/PCDF.

REFERENCES

Apostolopoulos, D., Marinakis, G., Ntantogian, C., Xenakis, C. (2013). Discovering authentication credentials in volatile memory of android mobile devices. Collaborative, Trusted and Privacy-Aware e/m-Services. *Springer Berlin Heidelberg*, p. 178-185.

Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P, Weisgerber.S.(2016) ARTist: The Android Runtime Instrumentation and Security Toolkit. *Cornell University Library*. arXiv:1607.06619.

Brezinski, D., Killalea, T. (2002). Guidelines for evidence collection and archiving. *RFC 3227. IETF*.

Carrier, B. D. (2003). Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers. *IJDE*, 1(4).

Drake,J.J., Lanier, Z., Mulliner, C., Fora, P. O., Ridley, S. A., Wicherski, G.(2014). Android hacker's handbook. *John Wiley & Sons*.

Google. *Android Open Source Project - AOSP*. Available: http://source.android.com.

Hilgers, C., Macht,H., Müller, T., Spreitzenbarth, N.(2014). Post-mortem memory analysis of cold-booted android devices. In: *IT Security Incident Management & IT Forensics (IMF), Eighth International Conference on. IEEE*. p. 62-75.

Høgset, E. S. (2015). Investigating the security issues surrounding usage of Ephemeral data within Android environments. *Master thesis. UiT The Arctic University of Norway*.

Ligh, M. H., Case, A., Levy, J., Walters, A.(2014). The art of memory forensics: detecting malware and threats in windows, linux, and mac memory. *John Wiley & Sons*.

Sabanal,P. (2014).*State Of The ART. Exploring The New Android KitKat Runtime*. https://conference.hitb.org/hitbsecconf2014ams/materials/D1T2-State-of-the-Art-Exploring-the-New-Android-KitKat-Runtime.pdf. Accessed October 20, 2016.

Sabanal,P. (2015). *Hiding Behind ART*. https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf. Accessed October 20, 2016.

Sylve J., Case, A., Marziale, L., Richard, G. G. (2012). Acquisition and analysis of volatile memory from Android devices. *Digital Investigation*, v. 8, n. 3, p. 175-184.

Wächter, P., Gruhn, M. (2015). Practicability study of android volatile memory forensic research. In: *Information Forensics and Security (WIFS), 2015 IEEE International Workshop on. IEEE*. p. 1-6.

Yaghmour, K. Embedded Android: Porting, Extending, and Customizing. *O'Reilly Media*, Inc.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS