DYNAMIC EXTRACTION OF DATA TYPES IN ANDROID'S DALVIK VIRTUAL MACHINE

Paulo R. Nunes de Souza, Pavel Gladyshev

Digital Forensics Investigation Research Laboratory,
University College Dublin, Ireland

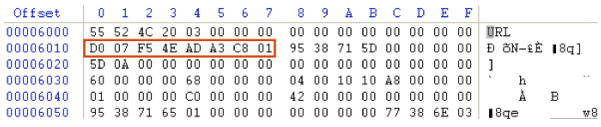
ABSTRACT

This paper describes a technique to acquire statistical information on the type of data object that goes into volatile memory. The technique was designed to run in Android devices and it was tested in an emulated Android environment. It consists in inserting code in the Dalvik interpreter forcing that, in execution time, every data that goes into memory is logged alongside with its type. At the end of our tests we produced Probability Distribution information that allowed us to collect important statistical information that made us distinguish memory values between references (Class, Exception, Object, String), Float and Integer types. The result showed this technique could be used to identify data objects of interest, in a emulated environment, assisting in interpretation of volatile memory evidence extracted from real devices.

Keywords: Android, Dalvik, memory analysis.

1. INTRODUCTION

In digital forensic investigations, it is sometimes necessary to analyse and interpret raw binary data fragments extracted from the system memory, pagefile, or unallocated disk space. Event if the precise data format is not known, the expert can often find useful information by looking for human readable ASCII strings, URLs, and easily identifiable binary data values such as Windows FILETIME timestamps and SIDs. Figure 1 shows an example of a memory dump, where a FILETIME timestamp can be easily seen (a sequence of 8 random binary values ending in 01). To date, the bulk of digital forensic research focused on Microsoft Windows platform, this paper describes a systematic experimental study to find (classes of) easily identifiable binary data values in Android platform.



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00006000	55	52	4C	20	03	00	00	00	00	00	00	00	00	00	00	00	URL
00006010	D0	07	F5	4E	AD	A3	C8	01	95	38	71	5D	00	00	00	00	Ð õN-£È ▮8q]
00006020	5D	0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	]
00006030	60	00	00	00	68	00	00	00	04	00	10	10	A8	00	00	00	` h ¨
00006040	01	00	00	00	C0	00	00	00	42	00	00	00	00	00	00	00	À B
00006050	95	38	71	65	01	00	00	00	00	00	00	00	77	38	6E	03	▮8qe w8

Figure 1: Hexadecimal view of a memory dump

2. BACKGROUND

Traditional digital forensics relies on evidences found in persistent storages. This is mainly due to the need to both sides of the litigation to reproduce and verify every forensic finding. The persistent storage can be forensically copied, providing a controllable way to repeat the analysis, getting to the same results.

An alternative way is to combine the traditional forensics with the so called live forensics. The live forensics relies on evidences found in volatile memory to draw conclusions. This type of evidence features a lesser level of control and repeatability if compared with traditional evidences. On the other hand, live evidences may unravel key information to the progress of a case. However, the question regarding the reliability of the live evidence remains in place, mainly in two moments: the memory acquisition and the memory analysis.

In the memory acquisition front, law enforcements and researchers are working to establish standard procedures to be used. These procedures could be based on physical or logical extraction. The physical extraction could need disassembling of the device or the use of JTAG as done by Breeuwsma

[2006]. The logical extraction can be more diverse, from interacting with the system with user privileges as done by Yen et al. [2009]; it could also gain system privileges through a kernel module as done by Sylve et al. [2012]; even use a virtual machine layer to have free access to the memory like done by Guangqi et al. [2014], among others. Regardless of the extraction method, there will be the need to analyse the extracted data.

One challenge faced when analysing a memory dump is that application data is stored in memory following the algorithms of the program owning that memory space. Being aware of the variety of software running on nowadays devices, the task of interpreting the device's extracted memory is complex. Some researchers are tackling this challenge taking different approaches. Volatility [2015] provides a customizable way to identify kernel data structures from memory dumps; Lin et al. [2011] used graph-based signatures to identify kernel data structures, Hilgers et al. [2014] uses the Volatility framework to identify structures beyond the kernel ones, identifying static classes in the Android system.

A deeper memory analysis tool that would consistently interpret data structures from application software has not yet being developed. The in-depth memory analysis is normally done in a adhoc basis, interpreting the memory dump from the light of the reversed engineered application's source code, as done by Lin [2011]. A broader approach, that would not depend on the application's source code, could be powerful to deep memory analysis.

This approach, not based on the application source code, would have advantages and disadvantages. As an advantage, this approach could be used in situations where the source code is unknown, unavailable, or legally disallowed to be reversed engineered. On the other hand, without the source code to deterministically assert the meaning of each memory cell, this method would need to take a probabilistic approach. The foundation for such approach is a probabilistic understanding of the memory data associated with their respective type. This paper uses the Android OS as environment to present a technique to gather the memory information associated with its type, making possible to have an probabilistic understanding of that data.

3. ANDROID STRUCTURE

The Android OS is an Operating System based on Linux, with extensions and modifications, maintained by Google. The OS was designed to run on a large variety of devices sharing same common characteristics [Ehringer, 2010]: (*1*) limited RAM; (*2*) little processing power; (*3*) no swap space; (*4*) powered by battery; (*5*) diverse hardware; (*6*) sandboxed application runtime.

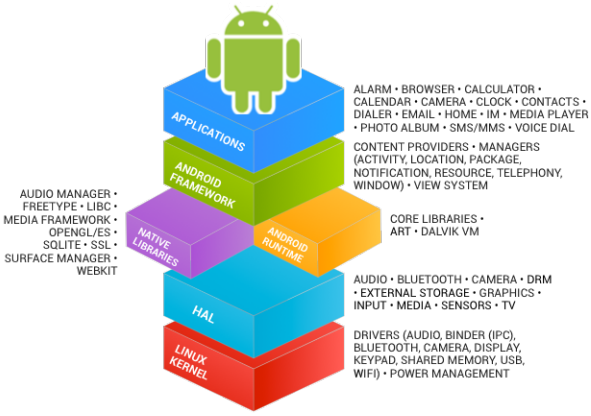


Figure 2: Architecture of Android OS

To provide a system that could run on such diverse and resource limited devices, they decided to build a multi-layered OS(Figure 2). The 5 layers are: (*1*) Linux kernel; (*2*) Hardware Abstraction Layer (HAL); (*3*) Android runtime and Native libraries; (*4*) Android framework; (*5*) Applications.

The Android OS is an hybrid of compiled and interpreted system. The boundary between compiled and interpreted execution is the Android runtime. The versions of the Android used in our experiments (*android-2.3.6 r1* and *android-4.3 _ r2.1*) feature Dalvik Virtual Machine (Dalvik VM) in the runtime package. All the programs running in the layers underneath Dalvik VM are compiled and all programs running in the layers above Dalvik VM are interpreted. The Dalvik VM hosts programs that were written in a Java syntax, compiled to an intermediary code level called bytecode and then packed to be loaded into Dalvik. When the software is launched inside Dalvik VM, each line of bytecode is interpreted into the machine code, normally in ARM architecture.

The Dalvik VM is implemented as a registerbased virtual machine. This mean that the instructions operate on virtual registers, being those virtual registers memory positions in the host device. The instruction set provided by the Dalvik VM consists of a maximum of 256 instructions, being some of them currently unused. Part of the used instructions is type specific, being those the ones chosen to be used to collect data and type information.

The Dalvik VM instruction set is grouped in some categories: **binop/lit8** is the set of binary operations receiving as one of the arguments a literal of 8 bits; **binop/lit16** is the set of binary operations receiving as one of the arguments a literal of 16 bits; **binop/2addr** is the set of binary operations with only two registers as arguments, being the result stored in the first register provided; **binop** is the set of binary operations with three registers as arguments, two source registers and one destination register; **unop** is the set of unary operations with two registers as arguments, one source register and one destination register; **staticop** is the set of operations that perform over static object fields; **instanceop** is the set of operations that perform over instance object fields; **arrayop** is the set of operations that perform over array fields; **cmpkind** is the set of operations that perform comparison between two floating point or long; **const** is the set of operations that move a given literal to a register; **move** is the set of operations that move the content of a register to another register.

Each of those categories has a number of instructions specifically designed to operate over some data type. The whole instruction set distinguishes 12 data types, namely: (*1*) Boolean; (*2*) Byte; (*3*) Char; (*4*) Class; (*5*) Double; (*6*) Exception; (*7*) Float; (*8*) Integer; (*9*) Long; (*10*) Object; (*11*) Short; (*12*) String.

4. MODULAR INTERPRETER (MTERP)

As the Android OS is open source, the source code of the OS [Google, 2015], including the Dalvik VM, is available to be downloaded and modified. By inspecting the Dalvik VM source code in details, it was possible to identify that the interpreter[2] would be a strong candidate to host the data collecting code. The features that most suit our needs are: (*1*) there is an different entry for each bytecode instruction, called opcode; (*2*) several of the opcodes of the Dalvik VM are type related. Therefore, it is a good point to place the code designed to collect the data, relating values and types that goes to memory.

Even though the Dalvik interpreter is conceptually the central point from where every single line of Dalvik bytecode should pass through, there is one exception. The Android OS features an optimization element called Just In Time (JIT) compilation that can bypass the Dalvik interpreter [Google, 2010]. The JIT compiler is designed to identify the most demanded tracks of code that run over the Dalvik VM. After identified, those tracks would be compiled and, next time they were demanded, the JIT would call the compiled track, instead of calling the interpreter. This way, the code we use to collect our data would not be executed and the collected data would not be accurate.

JIT configuration	# of instructions logged
WITH_JIT = true	2,676,540
WITH_JIT = false	3,643,739

Table 1: Number of instructions logged during the Android booting process

In our tests, the JIT compiler would skip, on average, 26.5% of the type bearing instructions during the Android booting process(Table 1). To avoid this source of error, it was necessary to deactivate the JIT compiler on our test Android OS. The Android system contains an environment variable *WITH_JIT* that is used to deploy an Android system with or without JIT. In order to deactivate the Just In Time compilation, we edited the makefile Android.mk[3] and forced the *WITH_JIT* to be set to *false*.

Having deactivated the JIT, it is necessary to insert the logging code into the interpreter. The interpreter source code is put together in a modular fashion, for this reason it is called modular interpreter (mterp). For each target architecture variant there will be a configuration file in the *mterp* folder[4]. The

[2] The interpreter is located on the following directory of the Android source tree: /android/dalvik/vm/mterp

[3] The Android.mk is located on the following directory of the Android source tree: /android/dalvik/vm

[4] The *mterp* folder is located on the following directory of the Android source tree: /android/dalvik/vm/mterp

configuration will define, for each Dalvik VM instruction, which version of ARM architecture will be used and where the corresponding source code is located. In order to log all the designed instructions, several ARM source code files, scattered in the *mterp* folder, will need to be edit accordingly, and any extra subroutine could be inserted in the file *footer.S*. After all the codes are edited, it is required to run a script called *rebuild.sh*, located in the *mterp* folder, that will deploy the interpreter[5]. Finally, the Android system, that will contain the modified interpreter, need to be built.

When executing the deployed Android OS, the data extraction takes place. The extracted data is stored in a single file with one entry per line as shown in Listing 1. The key information we can find in each entry are the two last columns, containing the type and the hexadecimal value stored in memory.

Listing 1: Unprocessed log sample

```
D(285:298)   Object    = <0x41a1fc68>
D(285:298)   Int       = <0x00034769>
D(285:298)   Object    = <0x41a1fc68>
D(285:298)   Int        = <0x00011db5>
D(285:298)   Byte      = <0x2f>
D(285:298)   Int       = <0x00000000>
D(285:298)   Int       = <0x0000002f>
D(285:298)   Char      = <0x2f>
```

Having this file, we process it to separate one data type on each file and exclude any extra information apart from the hexadecimal value, as depicted in the Figure 3.

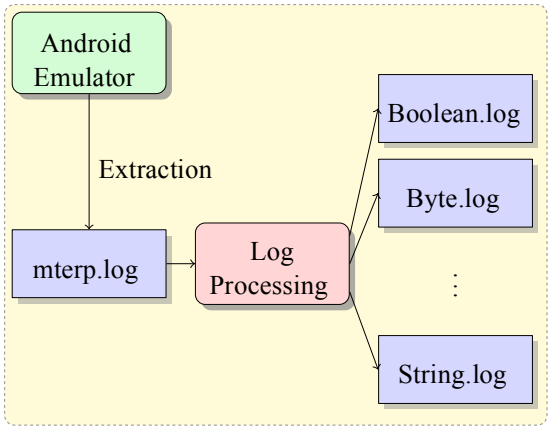


Figure 3: Log processing

Summing up, to extract the memory values associated with their respective types we needed to do the following:

- deactivate the JIT Compiler from an Android OS;
- inject code in the Dalvik Interpreter to log types and values on each interpreted typebearing instruction ;
- run the adjusted Android OS to collect data on the logs;
- process the logged data;

The deactivation of the JIT compiler and the modification in the Dalvik interpreted code, expectedly, generated an execution overhead. Considering the average booting time, the logging procedure seems to have effected more the response time than the JIT deactivation. The Table 2 shows the average booting times with and without JIT, as well as with and without the logging code.

	Log = off	Log = on
WITH_JIT = true	62s	2176s
WITH_JIT = false	62s	3026s

Table 2: Average booting time in seconds

5. RESULTS

Having all the processed logs, it was possible to extract some statistical information from them. The Table 3 shows in what proportion each type appear

[5] The interpreter is located on the following directory of the Android source tree: /android/dalvik/vm/mterp/out

in the logs. The table makes clear that the Int type prevail over the other types, with 54.3% of the appearances. Other types with a rather common rate of occurrence are Byte (8.17%), Char (13.19%) and Object (24.00%). The remainder of the types have a percentage lower than 1%.

Type	# of occurrences	% of total
Bool	6,512	0.1787%
Byte	297,578	8.1668%
Char	444,163	12.1898%
Class	1,454	0.0399%
Double	836	0.0229%
Exception	168	0.0046%
Float	6,374	0.1749%
Int	1,978,652	54.3028%
Long	7,837	0.2151%
Object	874,196	23.9917%
Short	3,034	0.0833%
String	22,935	0.6294%
Total	3,643,739	100.0000%

Table 3: Booting time in seconds

At this point, the 32-bit types are being highlighted. They are: (*1*) Class; (*2*) Exception; (*3*) Float; (*4*) Integer; (*5*) Object; (*6*) String. Each of those 6 types have its own probability distribution of values plotted on the Figure 4.

From the distributions it is possible to spot the similarity among the types: (*1*) Class; (*2*) Exception; (*3*) Object; (*4*) String. All 4 of them have a predominant peak a little after the value 0x4000000. This similarity can be explained by the fact that those 4 types are indeed references, therefore, pointers to a memory address. If focusing only on the values around 0x40000000, the Float type could be confused with the reference ones, because it also displays a peak around 0x40000000, however a much broader one, moreover, it has an second lower peak around 0xc0000000. The Int type displays occurrences along the whole spectrum of values, featuring two more relevant peaks. One peak around 0x00000000 and the other peak around 0xffffffff. Those two peaks could be explained by an greater occurrence of integer with small absolute values, being them of positive and negative signal, respectively.

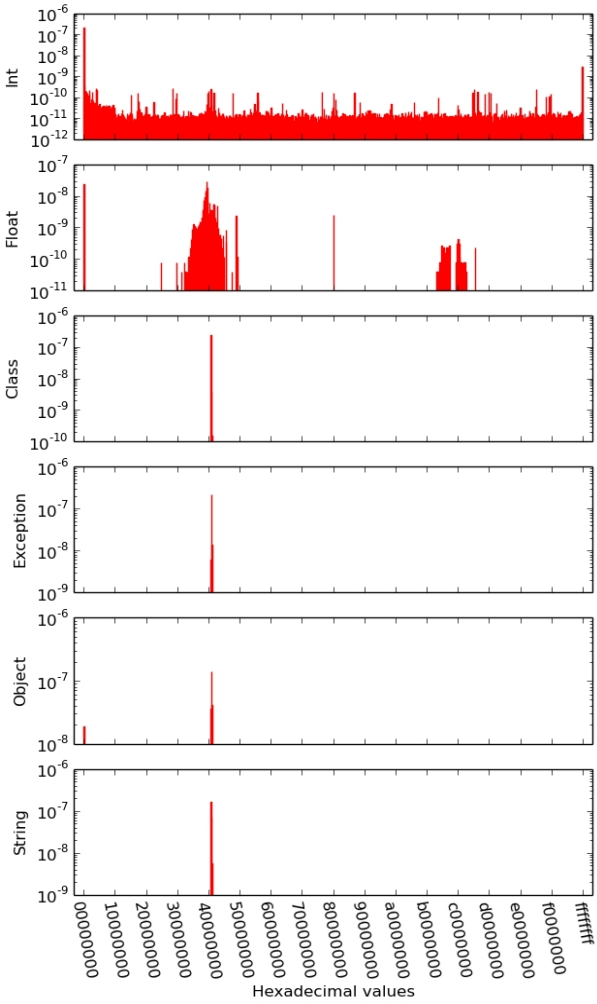


Figure 4: Probability distribution of values by 32-bit type (Log scale)

6. CONCLUSION

This paper explained a technique to capture memory data along with their corresponding data type in an emulated Android OS. This technique required deactivation of the optimization process called Just In Time compilation and the modification of the interpreter ARM code. The technique creates an expected overhead on the Android execution time. As this technique was only designed to run in emulated Android, this overhead is not an issue. The technique allowed us to collect important statistical information that made us distinguish memory values between references (Class, Exception, Object, String), Float and Integer

types. Beyond this specific test case, this technique could be use to build an statistical data corpus of Android memory content. This data corpus may become a tile on the work of paving the ground to the development of a consistent deep memory analysis tool.

7. ACKNOWLEDGEMENTS

This work was supported by research grants (BEX 9072/13-6) from Science Without Borders implemented by CAPES Foundation, an agency under the Ministry of Education of Brazil.

REFERENCES

Ing. M.F. Breeuwsma. Forensic imaging of embedded systems using JTAG (boundary-scan). *Digital Investigation*, 3 (1):32 – 42, 2006. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2006.01.003.

David Ehringer. The dalvik virtual machine architecture, 2010.

Google. Google i/o 2010 - a jit compiler for android's dalvik vm. Google Developers, May 2010. URL www.youtube.com/watch?v=Ls0tM-c4Vfo. Accessed 6th March 2015.

Google. Android source code repository. repo, 2015. URL https://android.googlesource.com/plataform/manifest. Accessed 11th February 2015.

Liu Guangqi, Wang Lianhai, Zhang Shuhui, Xu Shujiang, and Zhang Lei. Memory dump and forensic analysis based on virtual machine. In *Mechatronics and Automation (ICMA), 2014 IEEE International Conference on*, pages 1773–1777, Aug 2014. doi: 10.1109/ICMA.2014.6885969.

C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *IT Security Incident Management IT Forensics (IMF), 2014 Eighth International Conference on*, pages 62–75, May 2014. doi: 10.1109/IMF.2014.8.

Zhiqiang Lin. *Reverse Engineering of Data Structures from Binary*. PhD thesis, CERIAS, Purdue University, West Lafayette, Indiana, August 2011.

Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: brute force scanning of kernel data structure instances using graph-based signatures. In *18th Annual Network & Distributed System Security Symposium Proceedings*, 2011.

Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(34):175–184, 2012. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2011.10.003.

Volatility. The volatility framework, 2015. URL http://www.volatilityfoundation.org/. Accessed 18th March 2015.

Pei-Hua Yen, Chung-Huang Yang, and TaeNam Ahn. Design and implementation of a live-analysis digital forensic system. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, ICHIT '09, pages 239–243, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-662-5. doi: 10.1145/1644993.1645038.