SADFE 2015

Proceedings of the 10th International Conference on Systematic Approaches to Digital Forensic Engineering

Carsten Rudolph
Nicolai Kuntze
Barbara Endicott-Popovsky
Antonio Maña

UNIVERSIDAD DE MÁLAGA

Safe Society Labs

JDFSL The Journal of Digital Forensics, Security and Law

UFORIA - A FLEXIBLE VISUALISATION PLATFORM FOR DIGITAL FORENSICS AND E-DISCOVERY

Arnim Eijkhoudt & Sijmen Vos
Amsterdam University of Applied Sciences
Amsterdam, The Netherlands
a.eijkhoudt@hva.nl, sijmenvos@gmail.com

Adrie Stander
University of Cape Town
Cape Town, South Africa
adrie.stander@uct.ac.za

ABSTRACT

With the current growth of data in digital investigations, one solution for forensic investigators is to visualise the data for the detection of suspicious activity. However, this process can be complex and difficult to achieve, as there few tools available that are simple and can handle a wide variety of data types. This paper describes the development of a flexible platform, capable of visualising many different types of related data. The platform's back and front end can efficiently deal with large datasets, and support a wide range of MIME types that can be easily extended. The paper also describes the development of the visualisation front end, which offers flexible, easily understandable visualisations of many different kinds of data and data relationships.

Keywords: cyber-forensics, e-discovery, visualisation, cyber-security, computer forensics, digital forensics, big data, data mining

1. INTRODUCTION

With the growth of data that can be encountered in digital investigations, it has become difficult for investigators to analyse the data in the time available for an investigation. As stated by Teerlink & Erbacher (2006) "A great deal of time is wasted by analysts trying to interpret massive amounts of data that isn't correlated or meaningful without high levels of patience and tolerance for error".

Data visualisation might help to solve this problem, as the human brain is much faster at interpreting images than textual descriptions. The brain can also examine graphics in parallel, where it can only process text serially (Teerlink & Erbacher, 2006)

According to Garfinkel (2010), existing tools use the standard WIMP model (Window, Icon, Menu, Pointing device). This model is poorly suited to representing large amounts of forensic data in an efficient and intuitive way. Research must improve forensic tools to integrate visualisation with automated analysis, allowing investigators to interactively guide their investigations (Garfinkel, 2010).

Many computer forensic tools are not ideally suited for identifying correlations among data, or for the finding of and visually presenting groups of facts that were previously unknown or unnoticed. These limitations of digital forensic tools are similar to the forensic analysis of logs in network forensics. For example, logs residing in routers, webservers and web proxies are often manually examined, which is a time-consuming and error-prone process (Fei, 2007). Similar considerations apply to E-mail analysis as well.

Another issue with current tools is that they do not always scale well and will likely have problems dealing with the growth of data in digital investigations (Osborne, Turnbull, & Slay, 2010).

Currently, there are few affordable tools suited to and available for these use-cases or situations. Additionally, the available tools tend to be complex, requiring extensive training and configuration in order to be used efficiently.

Investigative data visualisation is used to assist viewers with little to no understanding of the subject matter, in order to reconstruct a crime or item and to understand what is being presented, for example an investigator which is not familiar with a particular scenario. On the other hand, analysis visualisations can be used to review data and to assess competing scenario hypotheses for investigators that do have an understanding of the subject matter (Schofield & Fowle, 2013).

A timeline is a valuable form of visualisation, as it greatly assists a digital forensic investigator in proving or disproving a hypothetical model proposed for the investigation. A timeline can also provide support for the mandate the digital forensic investigator received prior to commencing the investigation (Ieong, 2006). Interaction between role players can normally also be shown in network diagrams, so that the combination of a timeline and network diagram can generally answer many *who* and *when* answers.

The aspects of *what* and *where* can often be answered by examining the contents of evidence items, such as E-mails or the positional data of mobile phone calls. It is therefore important to be able to display the details of data with ease as well.

This paper describes the development of a flexible platform, Uforia (Universal Forensic Indexer and Analyser), that can be used to visualise many different types of data and data relations in an easy and fast way.

The platform consists of two sections, a back end and a front end, and is based on readily available open source technologies. The back end is used to pre-process the data in order to speed up the indexing and visualisation process handled by the front end. The resulting product is a simple and extremely flexible tool, which can be used for many types of data with little or no configuration. Very little training is needed to use Uforia, making it accessible and usable for forensic investigators without a background in digital investigations or systems, such as auditors.

2. ADVANTAGES

Uforia offers many advantages, of which the first is very low cost.

A second advantage is that the system scales well due to its use of multiprocessing and distributed technologies such as ElasticSearch, so that extremely large numbers of artefacts can be handled in a very short time. The processing of the Enron set, consisting of roughly 500 000 E-mails without attachments, typically takes less than ten minutes to complete on contemporary consumer-grade hardware. This pre-processing step also ensures that little to no processing needs to be done at the time of visualisation.

Thirdly, the Uforia's development heavily focused on making it as user- and developer-friendly as possible. Many forensic tools need a substantial amount of training and configuration to accomplish meaningful tasks. As this makes the systems difficult and expensive to use and develop for, it was considered paramount during Uforia's continued development to address these issues. Although a full UX study has not been conducted yet, the UI and feature set was developed using mock-ups and feedback from UX- and graphical designers, as well as potential users from several fields of expertise, such as process, compliance and risk auditors, forensic investigators and law enforcement officers, where none of the participants were given prior usage instructions.

Another advantage is the extreme flexibility of the system. It is very easy to add new modules, e.g. for handling new MIME types, as the programming of such a module can normally be accomplished in a very short time using simple Python programming. Additionally, the front end is completely web based, and no special software needs to be installed to use it. This, combined with the following common web design and UX standards, suggests that even novice users can achieve meaningful results with little to no training.

3. BACK END

3.1 START-UP PHASE

Uforia's back end is used to process the files containing the data that will eventually be indexed and used in the visualisation process.

The back end's first step is to create a MySQL table for the files. This table contains all metadata common to any digital file, as well as calculated metadata (such as NIST hashes).

A second database table is then generated, and it contains information about the supported MIME types. This table is built by looking at a configurable directory containing the modules for the MIME types that can be handled by the system.

Every module that can handle a specific MIME type is identified and added to this table. The table eventually contains zero, one or more 1:n key/value pairs for each of the supported MIME types and their respective module handlers. The module handlers are themselves stored as key/value pairs, with their original name as keys to the matching unique table name.

These tables are then created for each module, so that Uforia can store the returned, processed data from each particular module in its unique table.

Modules are self-contained files and extremely easy to develop. They only require the structure of their database table to be stored as a simple Python comment line in the particular module, starting with *# TABLE: ...*, and a predefined *process* function which should return the array of the data to be stored.

3.2 PROCESSING

Once all tables are created, the processing of the files that need to be analysed can start.

The first step is to build a list of the files involved. This is read from the config file. Once this list is completed, every file in the list is processed.

The MIME type of the file is determined and then the relevant processing modules (0, 1 ... n) are called to process the file. The results returned by each module are then stored in the database table that was generated earlier for that particular module.

When Uforia encounters a container format, it can deal with it efficiently by recursively calling itself. For instance, the Outlook PST module will unpack encountered PST files to a temporary directory and then call Uforia recursively for that temporary location. The unpacked individual E-mails are then automatically picked up by the normal E-mail module and processed accordingly.

Uforia can also deal efficiently with flat-file database(-like) formats by having modules return their results as a multi-dimensional array. Uforia's database engine turns these into multiple-row inserts into the appropriate modules' tables. Examples of modules that deal with flat-files in this fashion, are the modules that handle the mobile phone data (CSV-format) and the simple PCAP-file parser.

Due to its highly-threaded operation, the back end can pre-process large volumes of data efficiently in relatively little time. Once the processing steps are completed, the stored data needs to be transferred from the back end storage in JSON-format to the ElasticSearch engine for use by the visualisation front end.

4. FRONT END

The front end uses ElasticSearch, AngularJS and D3.js for the visualisation and administration interface.

The first step during the visualisation process is to select the modules or file types that need to be visualised in the admin interface.

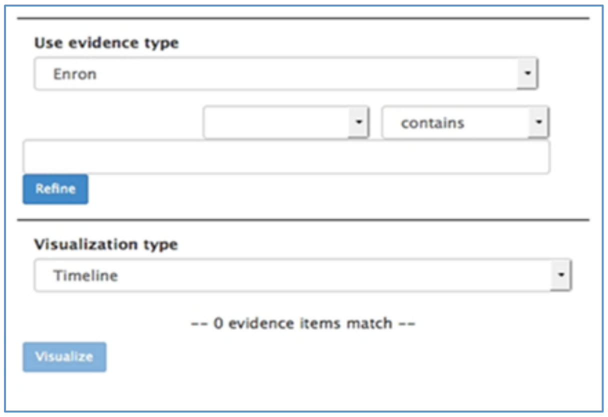
The next step is to select (and possibly group any identical) fields that need to be indexed by the ElasticSearch engine. The administration interface will hint at similar field names in other supported data types to allow for the merging of data types into one searchable set. This makes it possible to correlate the timing of e.g. cell phone calls and E-mails.

During or after the indexing and storing in ElasticSearch, one or more visualisations must then be assigned to the mapping in the admin interface. This also includes specifying the fields that should be laid out on the visualisation's axes.

The data in ElasticSearch can then be searched and visualised, even if the index process has not been completed yet. Because the front-end uses ElasticSearch, searches are fast and highly scalable. Only when full detail views of selected evidence items are necessary, the underlying back-end database needs to be accessed.

5. USER INTERFACE

The interface is designed with the goal of optimizing user-friendliness and ease of understanding. The user interface sports a 'responsive design', with UI elements automatically resizing and repositioning themselves for different screen sizes, such as with laptops, tablets and mobile phones, as can be seen in Figure 1.



Use evidence type
Enron
contains

Visualization type
Timeline

-- 0 evidence items match --

Figure 1: Mobile Interface

1) The user selects an 'evidence type', which is the name used for the collection, as it was generated in the admin interface
2) Uforia then loads the module fields that have been indexed for that evidence type, e.g. 'Content' for E-mails or documents.
3) The user selects whether the field should 'contain[s]' or 'omit[s]' the information in the last field.
4) Finally, the user selects one of the visualisations that have been assigned to the evidence type.
5) Uforia will now render the requested information using the selected visualisation, with some of the visualisations offering additional manipulation (such as a network graph).
   Lastly, all visualisations have one or more 'hot zones' where the user can 'click-through' to bring up a detailed view of the selected evidence item(s).

6. EXAMPLES

In this section, an examples can be seen of how Uforia is can be used to quickly determine the E-mail contacts of suspects. Despite limited available space in this paper, it is nevertheless possible to recreate similar scenarios for other data types.

Figure 2 shows an example set of a network graph derived from a sample set of PST-files, where the E-mail content was searched for the words 'investigate', 'books', 'suspect' or 'trading' and shown as a network graph indicating which individuals communicated about these words, with the size of the node indicating the amount of communication received. This immediately indicates the links between several possible suspects, including one whose PST mailbox was not included in the dataset and processed by Uforia.

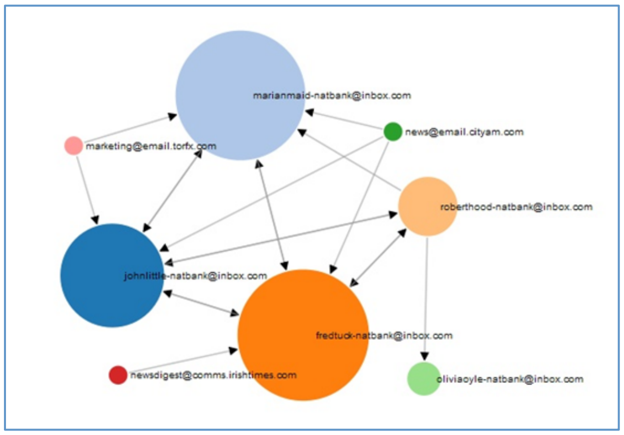

marianmaid-natbank@inbox.com
news@email.cityam.com
marketing@email.torfx.com
roberthood-natbank@inbox.com
johnlittle-natbank@inbox.com
fredtuck-natbank@inbox.com
newsdigest@comms.irishtimes.com
oliviaoyle-natbank@inbox.com

Figure 2: Network Graph

Another example is creating a timeline, as seen in Figure 3, to determine when messages were sent and which were sent around the time of the possible transgression.

It is easy to determine the times of the E-mail messages by hovering over the intersections on the timeline, and to investigate the original E-mails by clicking on the intersections (see Figure 4).

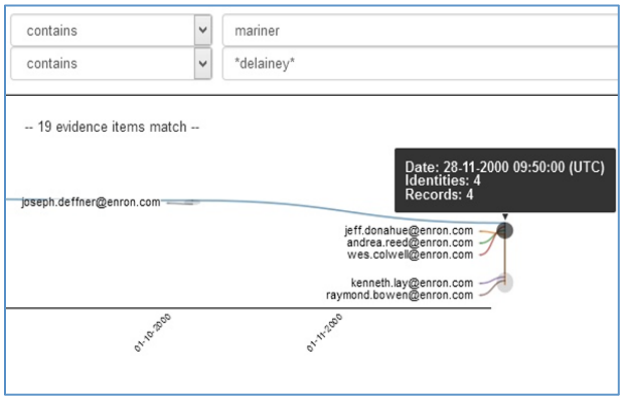

contains mariner
contains *delainey*
-- 19 evidence items match --
Date: 28-11-2000 09:50:00 (UTC)
Identities: 4
Records: 4

Figure 3: Timeline

The timeline visualisation can handle multiple items like calls from a large number of mobile phones. Figure 4 shows anonymised data from a real case, illustrating how contacts and time can easily be determined. The horizontal axis indicates the flow of time, while the graph nodes and coloured lines indicate the moment of contact between the two phone numbers. By clicking on the intersections, the original data can once again be displayed**.**

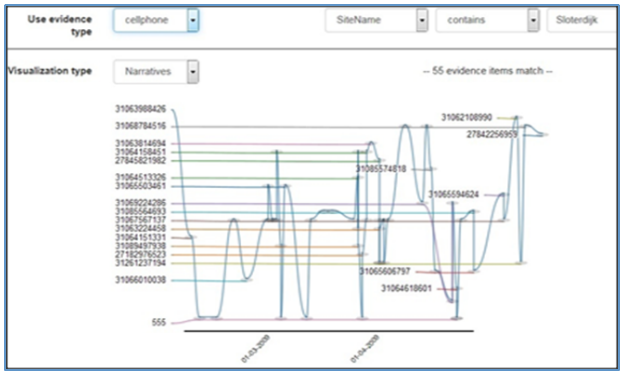

Use evidence type: cellphone
SiteName contains Sloterdijk
Visualization type: Narratives
-- 55 evidence items match --

Figure 4: Mobile Phone Timeline

7. CONCLUSION

Uforia shows that it is possible to create a simple, user-friendly product that is nevertheless powerful enough to use in the most demanding investigations.

It is easy to extend if any new MIME types are encountered or new features are needed.

Uforia was tested on a number of real life scenarios, and in all cases it was able to produce real results in a fast and efficient way, requiring hardly any operator training.

In conclusion, Uforia is fast, flexible and low cost solution for investigating large volumes of data.

REFERENCES

Fei, B. K. (2007). *Data Visualisation in Digital Forensics.* Pretoria, South Africa: Maters Dissertation, University of Pretoria.

Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. *Digital Investigation*, 64-73.

Ieong, R. S. (2006). FORZA - Digital forensics investigation Framework that incorporate legal issues. *Digital Investigation*(3), 29-34.

Osborne, G., Turnbull, B., & Slay, J. (2010). The 'Explore, Investigate and Correlate' (EIC) conceptual framework for digitalforensics Information Visualisation. *International Conference on Availability, Reliability and Security*, (pp. 630 - 634).

Schofield, D., & Fowle, K. (2013). Visualising Forensic Data : Evidence (Part 1). *Journal of Digital Forensics, Security and Law, Vol. 8(1)*, 73-90.

Teerlink, S., & Erbacher, R. F. (2006). Foundations for visual forensic analysis. *7th IEEE Workshop on Information Assurance.* Westpoint, NY: IEEE.

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

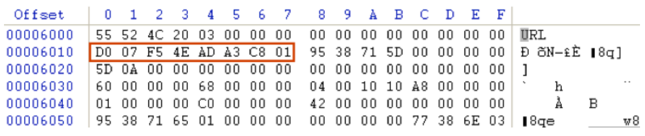
ABSTRACT

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.

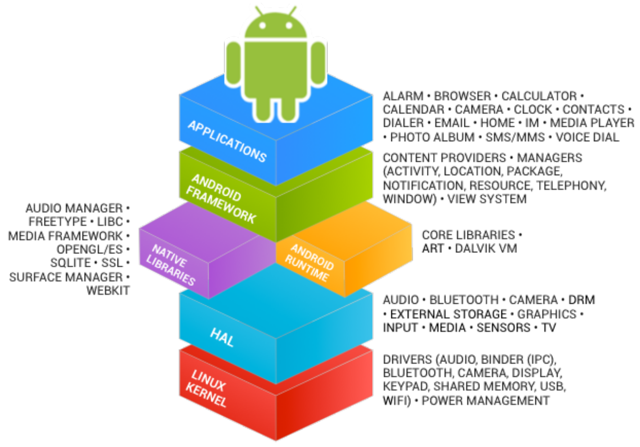

APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

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.

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

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

[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.

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

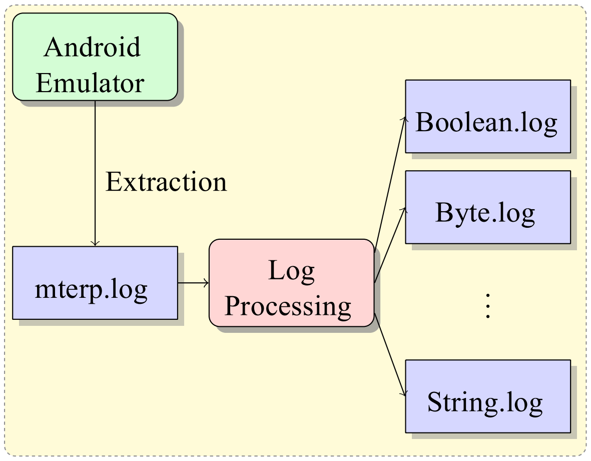


Figure 3: Log processing

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 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.

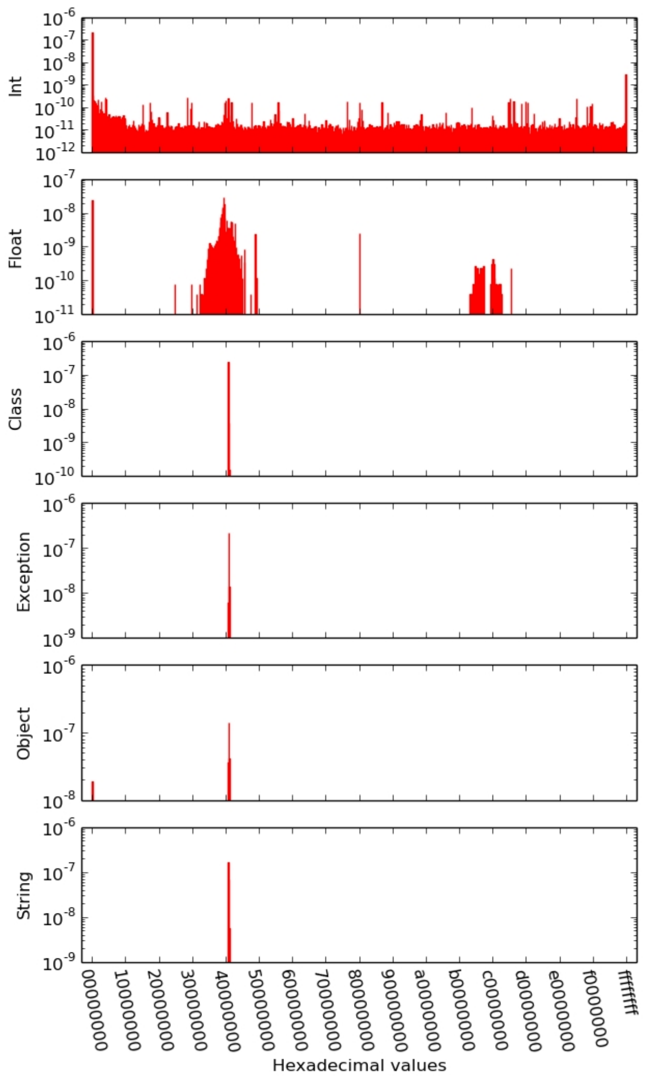


Figure 4: Probability distribution of values by 32-bit type (Log scale)

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.

Carsten Rudolph, Nicolai Kuntze,
Barbara Endicott-Popovsky, Antonio Maña

Proceedings of the 10th International Conference on Systematic Approaches to Digital Forensic Engineering

The SADFE series feature the different editions of the International Conference on Systematic Approaches to Digital Forensics Engineering. Now in its tenth edition, SADFE has established itself as the premier conference for researchers and practitioners working in Systematic Approaches to Digital Forensics Engineering.

SADFE 2015, the tenth international conference on Systematic Approaches to Digital Forensic Engineering was held in Malaga, Spain, September 30 – October 2, 2015.

Digital forensics engineering and the curation of digital collections in cultural institutions face pressing and overlapping challenges related to provenance, chain of custody, authenticity, integrity, and identity. The generation, analysis and sustainability of digital evidence require innovative methods, systems and practices, grounded in solid research and understanding of user needs. The term digital forensic readiness describes systems that are build to satisfy the needs for secure digital evidence.

SADFE 2015 investigates requirements for digital forensic readiness and methods, technologies, and building blocks for digital forensic engineering. Digital forensic at SADFE focuses on variety of goals, including criminal and corporate investigations, data records produced by calibrated devices, as well as documentation of individual and organizational activities. Another focus is on challenges brought in by globalization and cross-legislation digital applications. We believe digital forensic engineering is vital to security, the administration of justice and the evolution of culture.

ISBN: 978-84-608-2068-0

UNIVERSIDAD DE MÁLAGA

Safe Society Labs

JDFSL The Journal of Digital Forensics, Security and Law