DexGraph: a reverse-engineering static tool for Android binaries.

Maxence Ho
0ho.maxence0@gmail.com

Supervisor
Guillaume Bonfante, Loria

Abstract

Software analysis through reverse-engineering is the key process for anti-virus providers to detect and counter malware. The `Gorille` tool developed at Inria by Guillaume Bonfante's team uses morphological analysis based on Control-Graph-Flow (CFG) to provide automatic binary analysis and malware detection. Dynamic and static tools for building CFGs from binaries do exist for Windows, however this is not the case for Android binaries. This article describes and presents such a tool, `DexGraph`, able to build a CFG from an Android binary and to ouput it in various formats.

1. Source code

`DexGraph` source code and instructions can be found on Github:
`https://github.com/ChiminhTT/DexGraph`

2. Introduction

The work that is being described in this paper, supervised by Mr.Bonfante, is to be used directly by `Gorille`. Hence it might be interesting to start by presenting `Gorille` in order to understand what a direct application could be.

2.1. GORILLE: overview

Software analysis by security laboratories like Kaspersky or Symantec is typically done mostly manually by security expert engineers; the reason being that to reliably detect a malware one has to be able to understand the inner workings of said malware. The `Gorille` tool, that has been developed by Mr.Bonfante's team at Inria, is a software analysis tool that aims at speeding up this forensics process by automating a certain aspect of the analysis. In short it is able to indicate a degree of similarity between two given binaries [4, 5, 6, 7].

As a morphological analysis tool, `Gorille` relies on a particular representation of the software that is being analyzed; this representation is the control-flow-graph (CFG).

This graph is merely a representation of all paths that might be traversed through a program during its execution

Figure 1: Example of instruction-based CFG given by IDA disassembler

and is generally used by static analysis tools [3] and compilers (for compiler optimizations [1]).

To analyze Windows binaries, `Gorille` uses static and dynamic tools to get the instruction CFG. Once the CFG obtained, a certain number of modifications are performed on the CFG to remove as much as possible non-pertinent information and trivial schemes, giving what will be called the Abstract CFG. At that point the CFG is divided in small segments or Sites (usually about 20 nodes) that will be used to build a database representing the software being analyzed [4].

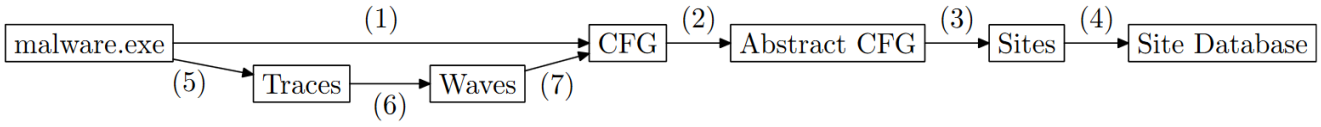

Figure 2: The `Gorille` data flow [4]

As with most malware detection tools, `Gorille` relies on a learning phase before being able to make comparisons. A malware database is given to `Gorille` allowing to build a database of all the malware's Sites (each Site being identified to its malware). Once this database is available, analyzing a binary consists in applying the same process to build the binary's Sites, which allows for statistical comparison to the database allowing to determine if this binary is close to a known malware.

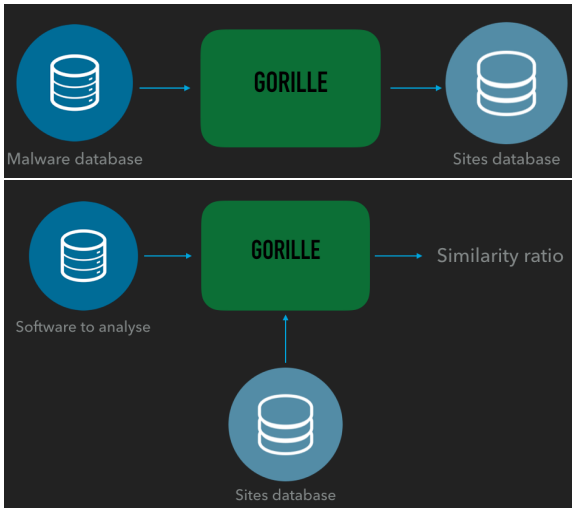

Figure 3: `Gorille` analysis process.

2.2. DexGraph original requirement

The actual implementation of `Gorille` starts at step (2) in Fig.2, allowing it to be platform agnostic. This means that it is expected to be fed the binary's CFG. Static and dynamic tools do exist (Traces, Waves[4]) to build this CFG from Windows binaries. However this is not the case for Android binaries.

The original requirement that was given was to find or develop a tool to obtain CFGs from Android binaries.

3. Analyzing Android binaries

Building an instruction-based binary's CFG requires to be able to access all the instructions contained within said binary file. A reverse-engineering task will be required to recover these information. A binary file, or in our case an executable, is a non-text file containing encoded instructions that can be interpreted by the computer's processor or GPU. Being almost always generated automatically from source code, binary files are very formated; however the formats are highly platform dependent (`ELF` for Linux, `Mach-O` for MacOS, `PE` for Windows).

The preliminary work to `DexGraph` was to find and analyze Android executable format.

3.1. Android runtime

The Android operating system is made of 4 layers:

- Java-based applications, executed in Dalvik-executable (Dex) code.
- System libraries, frameworks and services in Java.
- Low-level libraries (networking, graphic engine) in C/C++.
- The Android Linux kernel (by opposition of the 3 above that are located in the User Space)

The Android runtime is the part of the Android operating system that is in charge of executing applications. Originally, the android runtime was based on the Dalvik Virtual Machine (executing Dalvik based binaries). With *Android 2.2 "Froyo"* a `Just-In-Time` (JIT) compilation was added to the Dalvik virtual machine, allowing to dynamically profile applications and optimize instruction blocks on-the-fly. However with *Android 5.0 "Lollipop"*, the Dalvik virtual machine was ubiquitously replaced by the Android Runtime (ART) which enables `Ahead-Of-Time` (AOT) compilation - compiling as much of the application to native instructions as possible, allowing space and time performance gains. That said ART is backward-compatible with the Dalvik virtual machine as it requires the same input: the Dalvik-bytecodes contained within the standard .dex file inside the APK container (the APK container is what we commonly call the app)

3.2. Analyzing Dexfile format

Java being the main First-class language on Android for the moment, most application are written

in Java (or a Java derivate such as Kotlin). For the rest of the paper, we will assume that the binary being analyzed was originally written in Java.

Developers initially write Android applications in Java. The given code is then automatically transformed by the Java toolchain into bytecode after compilation. Java bytecode is an intermediate representation that is still agnostic of the specific virtual machine it is supposed to run on. Different types of Java virtual machine do exist from the proprietary Oracle Java virtual machine to the original Dalvik virtual machine for Android. The bytecode representation is optimized depending on the target virtual machine. For the Dalvik virtual machine, bytecode is transformed into a Dex representation: the DexFile contained within the Apk folder.

In a classic Java environment (Oracle Java virtual machine - JVM), Java code is compiled into `.class` files that are read at runtime by the JVM. Each Java class in the source code will correspond to a `.class` file after compilation. With the Dex format, all the information is contained within one and only file, the Dexfile (`classes.dex`). The structure of the Dexfile reflects the concerns for memory usage that it had to improve[2].

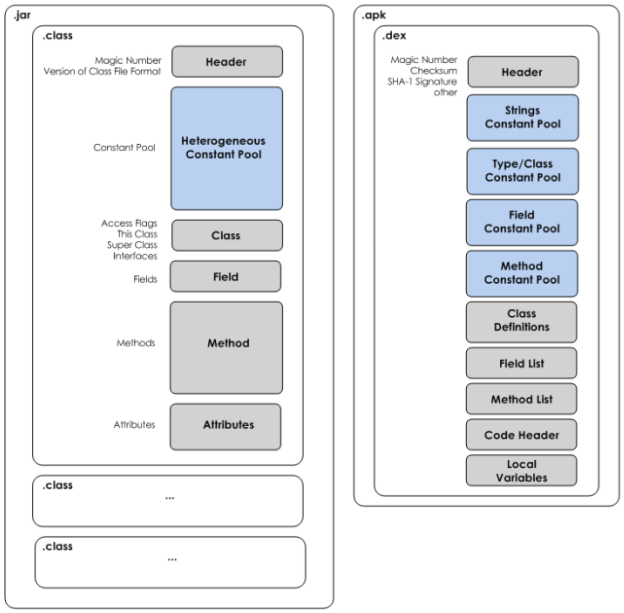

Figure 4: Side-by-side comparison of classic jar executable and Android's dex executable[2].

The structure of the Dexfile is described in Fig.4. This format was designed to avoid duplications. For example all the String literals needed to describe the program are exposed at the beginning of the file; if several methods need to use the same String (e.g "`Ljava/lang/String`") they will all refer to the id of this String in the "Strings Constant Pool".

```
public interface Zapper
{
  public String zap(String s, Object o);
}
public class Blort implements Zapper
{
  public String zap(String s, Object o)
  {
    ...;
  }
}
public class ZapUser
{
  public void useZap(Zapper z)
  {
    z.zap(...);
  }
}
```

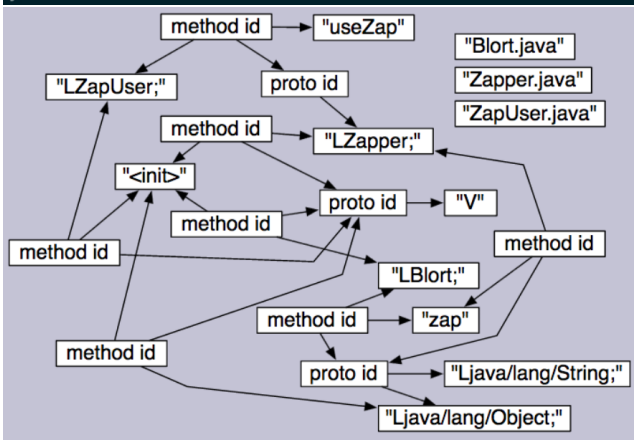

Figure 5: Java code excerpt and its dex representation.

3.3. Dexdump

The Android SDK platform provides a Google-developed disassembler tool, `DexDump`. This tool is able, from a *classes.dex* file, to output information on every class, every method inside the binary up to the list of instructions.

Fig.6 shows the type of information that `DexDump` is able to output. We can see that for every method in every class `DexDump` is able to indicate the set of instruction representing the method. Although the output is preformated and cannot be used as is, it contains every bit of information needed to build an instruction-based CFG as it exposes all the instructions in an ordered way.

Additionally `DexDump` being open-source, it is possi-

Figure 6: Example of `DexDump` output.

ble to analyze the tool's source code; and even better to iterate upon that base.

4. Building onto Dexdump: DexGraph

Since the "official" Google tool contained all the information required to build the CFG, the decision was made to use it as a starting point for `DexGraph`. At that point two options were possible to recover the information in an unformated way. Either parse `DexDump` output file and recover the instructions - but parsing of complex files is very tricky and particularly slow. Or, since the source code was available on a permissive Apache license, modify directly the source code to build `DexGraph`.

For evident reasons, the choice was made not to rely on parsing but to iterate upon `DexDump` source code.

4.1. Building the CFG for each method of the binary

`DexDump` acts on two levels. On a low level, it reads the binary file to get the information (according to the format described in Fig.4). On a high level, a basic algorithm (see Algo.1) to dump all the information of the binary (using the low-level output). Note that with this approach the program does not build an abstract representation of the whole file before dumping the information, which complicates any reconfiguration of the tool.

Algorithm 1 DexDump original algorithm

```
1: for class in dexfile do
2:     print(class.information)
3:     for method in class do
4:         print(method.information)
5:         for instruction in method do
6:             print(instruction.information)
```

At this point it is possible to obtain a somewhat ordered list of instructions for each method in each class. The first step to obtain the CFG is to divide this list into smaller segments corresponding to each of the segment/branch of the CFG: this implies detecting separating nodes. Branching in an instruction-based CFG depends on the opcode of the instruction: an `if` or `switch` instruction will always be followed by a fork. Once the list of segments obtained, we can use the extra information to connect them (for example connect the segment finishing by an `if` opcode with the segment corresponding to the *true* branch and the segment for the *false* branch).

Algorithm 2 Method CFG construction

```
1: procedure METHOD'S CFG CONSTRUCTION
2:     instr_vec ← list of method's instructions
3:     segments_vec ← get_segments(instr_vec)
4:     process_if_clusters(segments_vec)
5:     process_jmp_clusters(segments_vec)
6:     process_switch_clusters(segments_vec)
7:     return segments_vec
8: procedure GET_SEGMENT(instr_vec)
9:     segments_vec ← empty vector
10:    for instruction in instr_vec do
11:        if instruction.opcode is separator then
12:            s = truncate(instr_vec, upto: instruction))
13:            segments_vec.append(s)
       return segments_vec
```

Algo.2 describes on a high level the code that was implemented to create the intermediate CFG of a method, given its list of instructions, using a basic node/tree data structure. Fig.7 represents what would be stored in memory at that point, a small graph of the method with an entry and exit point, where each node represents an instruction (containing its Dalvik opcode and all the extra information that would be given by `DexDump`)

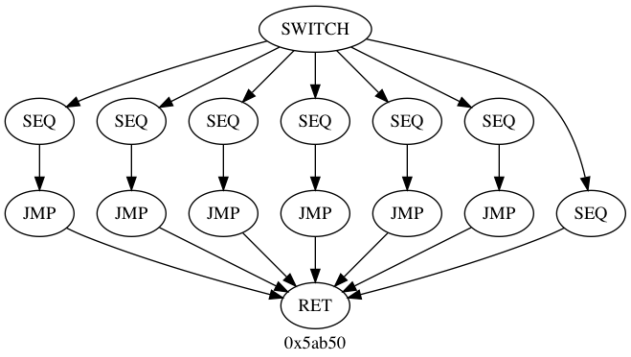

Figure 7: Dotfile representation of the method intermediate CFG.

4.2. Connecting the methods's CFGs

At that point, we dispose of the intermediate CFG for each method; intermediate because instructions with a `Call` opcode were not processed. The reason is that a `Call` instruction directly links to a given method, and to process it we would need to dispose of all the methods beforehand.

The next step is to process all `Call` instruction nodes by linking the corresponding method CFG. Algo.3 describes the approach that was taken to implement this step.

Algorithm 3 CFG Call processing

```
1: intermediate_cfg_vec ← empty vec
2: call_node_vec ← empty vec
3: for class in dexfile do
4:     for method in class do
5:         method_call_vec = get_method_call(method)
6:         method_inter_cfg = get_method_cfg(method)
7:         intermediate_cfg_vec.append(method_inter_cfg)
8:         call_node_vec.insert(method_call_vec)
9: process_call(intermediate_cfg_vec, call_node_vec)
```

4.3. Dumping the output

Once step 1 and 2 completed, we can store in memory a representation of the CFG. However this representation needs to be materialized to be usable. In fact, 2 output formats were required during `DexGraph` development: an in-house Edg format needed by `Gorille` as input and a Dot-file format needed to visualize a representation of the CFGs for debugging purposes.

Being able to transform a RAM representation of the CFG into a palpable output requires the implementation of an algorithm able to visit each node of the tree and apply a transforming method on each traversed node (see Algo.4 for a high level representation of the algorithm used).

Algorithm 4 CFG visiting algorithm

```
1: current_node ← root node
2: visited_node_set ← empty vec
3: visiting_node_stack ← empty vec
4: visiting_node_stack.emplace(current_node)
5: while visiting_node_stack is not empty do
6:     while current_node has child do
7:         visiting_node_stack.emplace(current_node.left_child)
8:     while visiting_node_stack.top.children < 2 do
9:         formating_method(visiting_node_stack.top)
10:        visited_node_set.emplace(visiting_node_stack.top)
11:        visiting_node_stack.pop()
12:    if visiting_node_stack is not empty then
13:        stack_top = visiting_node_stack.top
14:        current_node = stack_top.next_child
```

References

[1] C.Lattner. Llvm: An infrastructure for multi-stage optimization.

[2] D.Ehringer. The dalvik virtual machine architecture, 2010.

[3] D. M. T. F.Besson, T.Jensen. Model checking security properties of control flow graphs, 2005.

[4] F. G.Bonfante, J.Y Marion. Gorille sniffs code similarities, the case study of qwerty versus regin, 2015.

[5] F. A. G.Bonfante, J.Y Marion. Code synchronisation by morphological analysis, 2012.

[6] F. A. G.Bonfante, J.Y Marion. Analysis and diversion of duqu's driver, 2013.

[7] F. B. R. N. S. G.Bonfante, J.Y Marion. Regin en famille, une enqułte, 2014.