Rapid Android Parser for Investigating DEX files (RAPID)

Xiaolu Zhang [a, b], Frank Breitinger [b, *, 1], Ibrahim Baggili [b, 2]

[a] College of Computer Science and Technology, Jilin University, Qianjin Street 2699, Changchun, Jilin, 130012, PR China
[b] Cyber Forensics Research & Education Group, Tagliatela College of Engineering, ECECS, University of New Haven, 300 Boston Post Rd., West Haven, CT, 06516, USA

Introduction

With the wide adoption of the Android operating system, the number of Android applications on Google Play, the official Android Application market, is estimated to be over 1.5 million, a number which has steadily increased over the last ten years.[3] Complimenting this growth has been a stark increase in security threats attributed to Android applications.

An Android application is a single file in the Android Application Package (APK) format which is a compressed container (a zip file). Within that container, one may find (1) AndroidManifest.xml which holds essential data about the application that the Android system *must* read before it can run its code (2) at least one Android Virtual Machine Dalvik EXecutable (DEX) file which is the actual compiled application (we introduce its layout in Sections (DEX file layout and Related work)) additional data/resources like images, libraries, etc.

At the time of writing this paper, there were four common procedures for analyzing DEX files:

Smali: The most common procedure was to disassemble a DEX file into *smali* code[4] which is based on Jasmin syntax[5] and is usually saved in text format. Next, these text files (one per class) can be parsed and aid in further analysis.

* Corresponding author.
E-mail addresses: xiaoluzhang1985@gmail.com (X. Zhang), FBreitinger@newhaven.edu (F. Breitinger), IBaggili@newhaven.edu (I. Baggili).

[1] http://www.FBreitinger.de/.

[2] http://www.unhcfreg.com/.

[3] http://www.appbrain.com/stats/number-of-android-apps (last accessed Dec. 6, 2015).

[4] Smali code is a human-readable representation for Dalvik bytecode.

[5] http://jasmin.sourceforge.net/guide.html (last accessed Dec. 6, 2015).

DEX2JAVA: The second possibility was converting DEX files into JAVA bytecode which results in either a .jar file or several .class files. This allows utilizing already existing tools for JAVA bytecode analysis.

Manual analysis: While the first two approaches are automated, a third method is to employ an interactive tool (a debugger or disassembler like IDA Pro) and work directly on the DEX file.

Individual solutions: Some researchers implemented their own standalone programs for parsing/disassembling Android applications. This is discussed further in Section (Related work).

While the aforementioned procedures are currently common, there are several disadvantages (depending on the procedure): (i) one has to be familiar with the smali syntax (ii) The first two procedures employ an intermediate format which is time consuming and requires more disk space (iii) the conversion from DEX to JAVA is not reliable and several applications cannot be converted causing converters to crash (iv) the offset/location of the data extracted from the intermediate file(s) is difficult to acquire from a forensic examiners' perspective since the intermediate representation cannot explicitly link where it is acquired from in a DEX file and (v) the 'manual' procedure is only appropriate for a small number of applications as it requires a practitioner to manually extract and analyze relevant data.

Given these limitations, this paper presents Rapid Android Parser for Investigating DEX files (*RAPID*), an open source tool for DEX file analysis that is efficient (runtime), can handle large amounts of data, and is easy-to-use for forensic practitioners due to its well-documented APIs (Github plus javadoc).

The performance improvement in our method is gained by directly working on the DEX files. Furthermore, the devised RAPID approach does not require additional storage space. Lastly, examiners do not have to be familiar with any intermediate syntax (e.g., smali). In case an application requires additional, manual inspection, RAPID provides the exact offset of the data acquired (e.g., where a string is stored inside the application).

Additionally, there are two other advantages to RAPID. Primarily, in our experiments, we obtained errors when decompiling/converting DEX files with traditional tools which did not happen with RAPID (see Section (Reliability)). Second, RAPID can support dynamic analysis and guide examiners to suspicious offsets[6] (see Section (Use case: finding outsourced functionality)).

The results show that for our sample set of $n = 11,711$ Android applications, 16 applications could not be decompiled/converted with existing tools, while RAPID handled them correctly. Furthermore, for the remaining 11,695 samples with a total DEX file size of 22.35 GB, RAPID reduces the query time from 1368 min–88 min.

The rest of this paper is organized as follows: Section (DEX file layout) summarizes the DEX file layout followed by the related work in Section (Related work). The core of this article is Section (RAPID approach) which describes the approach, the implementation, some details about the parsing, the usage including APIs, a special use case as well as the validation. The sec: experiment section discusses RAPID's benefits. The last two sections provide the limitations followed by the conclusions and future work.

DEX file layout

The DEX file structure is well-documented on the official Android Dalvik Executable format page (Google, 2008). An overview is provided in Fig. 1 where the left side shows a high-level synopsis similar to the official documentation.[7] On the right hand side we present a slightly more detailed representation of the data section which RAPID utilizes to parse DEX files.

A DEX file is made up of several sections where Fig. 1 outlines the most important ones (with respect to application analysis). The starting point is usually the *header* which provides pointers to the other major sections. Focusing on the actual content, *string_ids* and *string_data* contain all the data about strings. 'String' here refers to the parts of operations and definitions which have to be represented by string labels (e.g., value of string constants, type and class names etc.). The *method_ids* section contains indexes leading to data related to methods, e.g., which class they belong to, method names, type of parameters etc. The *code* section comprises all code instructions divided by code blocks referring to the methods defined in a DEX file. More details are presented in Section (Parsing a DEX file) where parsing is elaborated on.

Related work

The introduction including Section (DEX file layout) briefly outlined the structure and layout of Android applications. In this section, we discuss disassemblers followed by work relevant to APK file analysis.

Commonly, Android applications are investigated by analyzing the AndroidManifest.xml, the DEX file or both (Talha et al., 2015). The XML-file processing is straightforward – convert binary into text and parse it. Since XML-files are usually small in size, this process is quite easy and efficient. However, a DEX file is more challenging as it can be larger in comparison.

To the best of our knowledge, there are currently 12 tools for analyzing DEX files. An overview of these tools is presented in Table 1 with the name of each tool including a link to their websites. The third column contains a short description followed by some literature that utilizes each of the mentioned tools.

Rows 1–2 show works that decompile the DEX file into smali code using Baksmali or ApkTool. Smali/Baksmali is a prominent assembler/disassembler for DEX files that

[6] We can locate external function calls such as native libraries (*.SO files) or JAVA executable files (DEX, JAR). This technique can be used to hide / obfuscate code.

[7] https://source.android.com/devices/tech/dalvik/dex-format.html#file-layout (last accessed Dec. 6, 2015).

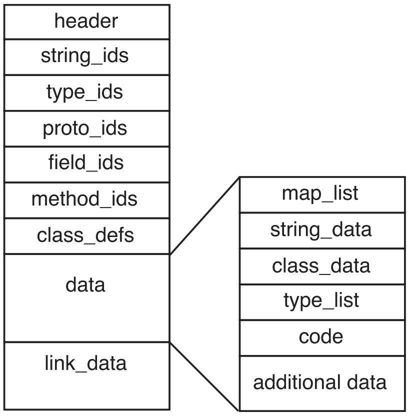

Fig. 1. DEX file layout overview.

Table 1
Works regarding to DEX file analysis.

	Tool	Description	Utilization
Smali			
1	ApkTool[a,b]	Decompiles APK file	Wu et al. (2012), Zheng et al. (2012), Hoffmann et al. (2013)
2	Baksmali[a,c]	disassembles DEX file to smali files	Zhou et al. (2012), Lu et al. (2012)
DEX2JAVA			
3	Dex2jar[a,d]	converts DEX file to JAR file	Gibler et al. (2012)
4	Ded[a,e]	converts DEX file to.class files	Yang et al. (2013)
5	Dare[a,f]	converts DEX file to .class files	Elish et al. (2015)
Manual analysis			
6	Androguard[a,g]	reverse engineering APK file	Desnos and Gueguen (2011)
7	IDA Pro[h]	reverse engineering a wide range of binaries	Drake et al. (2014)
8	JEB[i]	reverse engineering APK file	Dmitrienko et al. (2014)
Individual solution			
9	AIS	disassembles DEX file to smali code	Zheng et al. (2013)
10	Own tool	converts DEX file to JAVA bytecode	Chen et al. (2013)
11	Own parser	parses DEX file for APIs and strings	Arp et al. (2014)
12	Dedexer[a,j]	disassembles DEX file to its own format	Chin et al. (2011), Seo et al. (2014)

For more details, please visit their own website.
[a] These tools are open source tools under different licenses, e.g., Apache 2.0, GPLv2, BSD 3-Clause, etc.
[b] https://ibotpeaches.github.io/Apktool/.
[c] https://code.google.com/p/smali/.
[d] http://sourceforge.net/projects/dex2jar/.
[e] http://siis.cse.psu.edu/ded/.
[f] http://siis.cse.psu.edu/dare/index.html.
[g] https://code.google.com/archive/p/androguard/.
[h] https://www.hex-rays.com/products/ida/.
[i] https://www.pnfsoftware.com/.
[j] http://sourceforge.net/projects/dedexer/.

outputs smali code. A positive aspect of this technique is that it fully supports the DEX format and also allows one to extract annotations, debugging information, and line numbers. The second open source tool, ApkTool, is a Smali/Baksmali decompiler/compiler for Android APK files. ApkTool has the ability to debug smali code step by step, and can build a language pack by translating the .xml strings inside APK files. While these tools are widely adopted, they come with a major downside of converting the binary code into smali code which is time consuming.

Rows 3–5 present the DEX2JAVA applications. Dex2jar, ded and dare (note, dare is the successor of ded) can convert the DEX files into JAVA bytecode (.jar, .class) and thus they convert from binary into binary (Enck et al., 2011). The benefit of this conversion is that there are already several existing tools for JAVA bytecode analysis which may then be utilized, e.g., Soot,[8] Jad[9] and JD-GUI.[10] Note, these tools can be used to process JAR files and therefore are not listed in Table 1 nor are they discussed. Notwithstanding, even though DEX2JAVA tools offer speed efficiency due to intermediate representation, Castillo et al. (2011) points out that the DEX2JAVA conversion is not reliable and often fails. For example, Yang et al. (2013) indicated that 42 out of 1750 samples resulted in a failure during their work using ded.

Rows 9–11 exemplify tools in the 'manual' category. Androguard allows decompiling and disassembling Android applications and is helpful when manually analyzing applications (Desnos, 2013). It is also a toolset for reverse engineering Android applications with the goal of malicious application detection, built into Santoku Linux[11]. On the other hand, one may use more general tools like IDA Pro which is a commercial tool for Windows, Linux and Mac OS X for application analysis. It is a multi-processor disassembler and debugger that offers many features, and can provide safe analysis of potentially harmful programs (Hex-Rays, 2005). Incidentally, JEB is another commercial interactive decompiler that is able to process multiple APK files to smali or JAVA source consecutively.

Rows 12–15 summarize works having an intermediate phase that implemented their own disassembler/parser that may generate a non-standard intermediate format. While this may be efficient as it is optimized for a specific purpose, it also means that researchers and practitioners reinvent the wheel as they have to develop a variety of parsers to acquire data from Android applications.

RAPID approach

As shown by our literature review, most works are based on gaining access to the data in APK files, and more importantly in DEX files. It is therefore critical for future work to adopt a more efficient, standardized, optimized and accurate approach for acquiring desired data from DEX files. Our solution to this problem is RAPID. This *in-memory* solution hinges on three major steps:

[8] http://sable.github.io/soot/ (last accessed Dec. 6, 2015).
[9] http://varaneckas.com/jad/ (last accessed Dec. 6, 2015).
[10] https://github.com/java-decompiler/jd-gui (last accessed Dec. 6, 2015).
[11] https://santoku-linux.com (last accessed Dec. 6, 2015).

Decompress: APK files equal zip files and thus the first required step is decompression which reveals the DEX files as well as the AndroidManifest.xml file. While DEX files serve as input for RAPID, AndroidManifest.xml is only converted into human readable text and is currently not required by RAPID.

Load DEX files: After decompressing, the data is pulled from the DEX files and is loaded into an internal data structure which consists of four main *components*: string, method, codeBlock and instruction. All queries and further processing are performed on this internal structure which resides in memory. More details on the internal data structure as well as the parsing are discussed in Sections (Implementation and Parsing a DEX file) respectively.

Query: Once the in-memory data structure is prepared, RAPID allows different query types (based on the *components*). For instance, investigators can look for a specific 'string', 'method-name', 'used APIs' or even 'find the exact offset in the code'. A detailed explanation of what data the queries in RAPID is able to return can be found in Section (Implementation).

Implementation

Our JAVA prototype implementation is open source and can be downloaded from https://github.com/unhcfreg/RAPID. RAPID comes in a form of a library (i.e., a JAR file), a sample.java which demonstrates some use cases with detailed documentation (generated with javadoc). Note, RAPID was compiled with JAVA 7 and thus requires JRE 7 or higher.

The implementation consists of four main components – string, method, codeBlock and instruction – where each component contains corresponding objects. For instance, the method component includes a list of method objects (one per method). The structure of each object, which are also the searchable fields is outlined in Table 2; a brief summary is provided in the following paragraphs (parsing level is described at the end of Section (Parsing a DEX file)):

String objects represent all strings that exist in the application and delineates the string_data section from the DEX file. This includes values in string variables, function/class names and function return values (e.g., void, int).

Method objects contain the data about specific methods. For instance, a method object knows its name, the class it belongs to, number and types of parameters and the return value (e.g., void) and associated executable code.[12]

CodeBlock objects link the methods to the actual instructions (bytecode). Therefore, a codeBlock has a start address (offset), end address (offset) and an instruction list (e.g., string-const v0 "Hello World" etc.).

Instruction objects embody the actual code that is executed and are necessary for flow analysis. For instance, it allows one to locate where specific methods were called from.

Note, method, codeBlock and instruction are linked to each other (methodId and the ArrayList<Instruction>) whereas the strings are duplicated and also stored in the corresponding objects, e.g., methodName can be found in a method object as well as in the string component. This was implemented for performance reasons.

Table 2
Summary of the main components and their attributes.

Type	Field	Description
String object (StringElement.java)		Parsing Level 1
int	stringId	index of the string from string_ids
long	address	offset pointing to the string content in the DEX file
String	stringContent	the string itself
long	stringLength	length of the string
Method object (MethodElement.java)		Parsing Level 2
int	methodId	index of the method from method_ids
long	address	offset pointing to the meta data of the method (in method_ids)
String	className	class name as string where the method belongs to
String	methodName	name of the method name
String[]	parameterType	type(s) of the parameter(s) of the method
String	returnValueType	type of the return value
boolean	hasCodeBlock[a]	true if the method is implemented in DEX file
CodeBlock	codeBlock[a]	pointer of the codeBlock
CodeBlock object (CodeBlock.java)		Parsing Level 3
int	codeBlockId	index number of the code block to link it to a method
long	startAddress	start offset of the code block in DEX file
long	endAddress	end offset of the code block in DEX file
int	methodId	index of the method the code block belongs to
ArrayList <Instruction>	instructionList[a]	list of pointers to instructions
Instruction object (Instruction.java)		Parsing Level 4
int	instructionId	index number of instruction to link it to the CodeBlock
int	codeBlockId	index number of the code block the instruction belongs to
long	address	offset of the instruction in DEX file
boolean	hasOperand	true if an instruction has an operand
boolean	hasRegister	true if an instruction has register(s)
int	length	length of instruction in bytes
int	op	hex value of the opcode
String	opcode	general name for same type of mnemonics
String	opcodeSuffix	specific mnemonic of opcode
long	operand	value of the operand
String	operandSuffix	explanation of operand value
int[]	registerList	list of registers

NULL for parsing level 2 and will be filled in parsing level 3.
[a] Are special fields which fall into the next parsing level, e.g., hasCodeBlock is an attribute of the method object.

[12] Note, some values (e.g., void) are redundant and can be found in string objects as well as in method objects. We decided for that due to the performance increase.

Parsing a DEX file

Parsing the DEX file is a complex task as it involves running through the bytecode and selecting relevant data. This section provides a short overview of the parsing process.

Although the DEX file format is well described by the Google (2008) documentation, we decided to include this overview as digital forensic practitioners and researchers continue to face issues in malware investigations due to the lack of the ability of tracing certain data by traversing contents of a DEX file.

Fig. 2 provides an overview of the complete parsing procedure and is explained in the subsequent paragraphs. As stated in Section (DEX file layout), the starting point is the *header* which contains pointers to the main sections.

First, the stringObjects are built where RAPID parses *string_ids* and then reads the data (1+2). Here, *string_ids* are pointers to specific strings. Second, RAPID works on the methodObjects where it starts at the *method_ids* (3) which allows it to acquire the method name from *string_data* (4+5). Note, since *string_data* is already parsed, we can retrieve this data from our stringObjects.

Next, RAPID reads the 'string ID for the class name' from *type_ids* which is then used to get the actual class name as a string (6–8). *proto_ids* contains two IDs – the return value ID and the pointer for the parameter list of the method. Hence, RAPID acquires the return type from *type_ids* and resolves it further into a string (10–12). Furthermore, it analyzes the *type_list* which contains data about the number of parameters as well as the all parameter types (13). The *type_ids* can be matched to names by parsing the corresponding sections (14–16).

Having the string and method object in memory, the final steps focus on creating the codeObjects which is executed by parsing steps 17–20. Note, instructions are part of the codeObjects. *Star-** is representative of all instructions as presented in Table 2,[13].

This parsing procedure allows for different parsing levels (see Table 2). That is, only required sections are parsed where lower levels always need to be parsed first. For instance, if the analysis only requires a string search, RAPID only creates the stringObjects, i.e., parsing level 1. If the search involves parts from the methodObjects, RAPID parses levels 1 and 2.

Usage

This section provides step by step instructions on how to install and run RAPID. It is meant to ease the usage process for potential practitioners.

Step1: Ensure that JAVA Development Kit (JDK) 1.7 or higher is installed.
Step2: Download the RAPID JAR library, sample.java as well as sample APK files and store them in the same directory. By default, the code will analyze all APKs that are in the same directory.
Step3: Compile the sample.java file in the system terminal by using the following command:

```
javac -cp RAPIDv0.2.jar sample.java.
```

Step4: Execute the sample.class file with the command

```
java -cp .; RAPIDv0.2.jar sample (on Windows)
java -cp .:RAPIDv0.2.jar sample (on *nix).
```

The output of the sample file presents a general overview of the DEX files such as the total number of strings, methods and APIs used in the application. Additionally, it prints the first 20 strings in the string component and the first 20 APIs with their basic data such as class and function name, address etc. Next, we chose a known JAVA API: java.lang.System.load(..) to test for its existence. If the result is (true), all the instructions invoking the API will be printed, of which the most important data is the address(es) where the API was invoked in the DEX file. Furthermore, the methods and the details of the codeBlock, where the instructions executed will be listed as well.

In total, RAPID v0.2 currently provides 27 APIs which are listed Appendix A including a short description for each one. These 27 APIs can be divided into four categories. The four 'setting'-APIs allow for initializing RAPID, e.g., setting the source directory of the APK samples. The second set of APIs contain the three 'main object queries'; functions of RAPID which return lists of the three main objects of the internal data structure: String, method and codeBlock (see Section (Parsing a DEX file)). The third set of APIs allow for specific queries against the complete data structure. A user can search for the existence of a string, method or API, or acquire a list of all classes. Those APIs are summarized in the 'search operations' section.

The last set 'Workflow analyses functions' include functionality to further inspect a given DEX file.

For instance, getMethodInvoker(..) can back trace the methods invoking a specific function as well as getExternalFilesDirs() can obtain where the external files are located.

The decision for these APIs was driven by existing literature; we analyzed what features/functions are required by existing tools and implemented those. For example, the malware detection concepts proposed by Wu et al. (2012) and Peiravian and Zhu (2013) utilize API calls only as their features, thus RAPID provides a method getApiList(). A detailed discussion about all of the APIs is beyond the scope of this article. For more details, readers may want to explore the documentation which comes with the RAPID library.

Although these 27 APIs allow access to most of the data stored within the data structure, there might be scenarios where different outputs are required. In that case one may have to implement their own logic and use the existing 'getter-'methods of the different objects.

Use case: finding outsourced functionality

A common problem when analyzing applications is outsourced code; developers have the option to place code/

[13] The actual parsing for * is complex and explaining it in detail is beyond the scope of this paper. We plan on publishing a technical report that outlines the exact procedure.

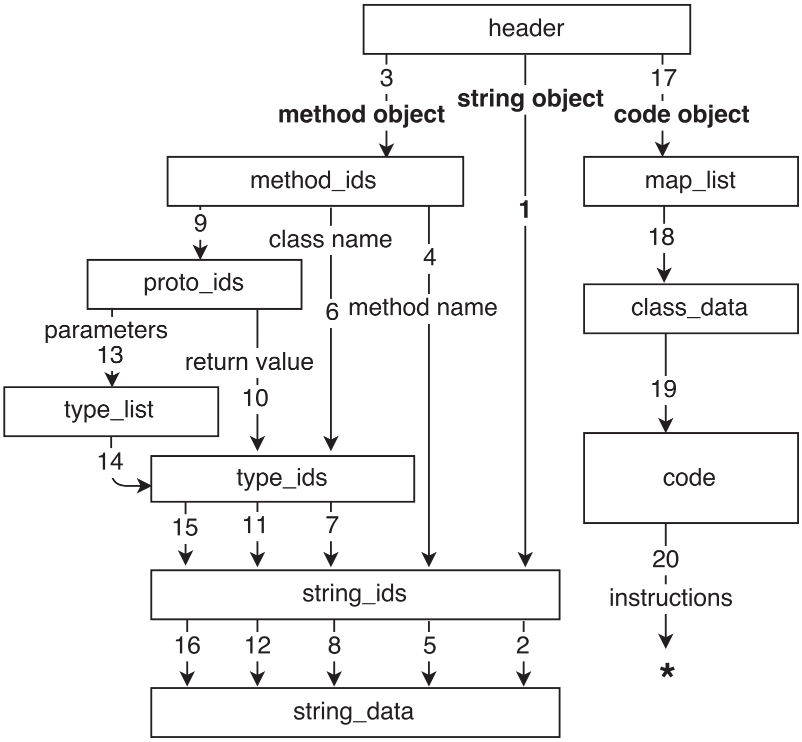

Fig. 2. Overview of the parsing procedure.

functionality in files other than the main DEX file. For instance, placing API calls or other functionality externally is sometimes used for obfuscation (Apvrille and Nigam, 2014). Thus, for investigative purposes, it is of interest to know if external files are being loaded.

External files and calls. There are two ways for an Android application to load code from external files, static and dynamic. The static method imports libraries or Java Archive (JAR) files into the program before the APK file is compiled. On the other hand, the dynamic procedure calls the external files/functions during runtime. Since static can be easily identified by checking the application's directory, we focus on dynamic loading.

In general, applications can load two types of external files: *.SO files or JAR/DEX files. SO files are native libraries following the Java Native Interface (JNI) standards which are developed by the Native Development Kit (NDK) and are usually written in C or C++.

JAVA provides four different APIs to load content dynamically. load(..) and loadLibrary(..) from class java.lang.System can load native libraries while the constructor of class dalvik.system.DexClassLoader and dalvik.system.PathClassLoader[14] are utilized for loading classes from DEX or JAR files.

The search process. Determining if an application calls any code from external files requires searching for the four APIs in the main DEX file. The procedure is generally divided into three steps:

Step1: Search if one of the APIs is invoked in a DEX file which can be performed by analyzing the instruction objects (in RAPID) or examining if it is part of a function.
Step2: Next, once the API is identified, the parameters are analyzed to explore whether we can figure out the library or the path to the library. For instance, if a library is dynamically loaded, it might be the case that the string already exists in the DEX file. If the string cannot be found, then we continue to step 3.
Step3: Obviously our approach does not perform dynamic analysis, however, this procedure provides the exact address of the invoke and thus a researcher or practitioner can use the acquired address and set the ‘break point’ during dynamic analysis.

To simplify this process, RAPID provides two APIs. The areExternalFilesLoaded() is a boolean function to test weather one of those four APIs was found. The second function name getExternalFilesDirs() returns a list of <key,value> objects where key is the address of an invoke and value is the actual name of the loaded lib/file. An example of the output of this function is shown below.

```
176088—>
176032—>
229790—>/system/lib/libandroid.so
```

The output shown means that three offsets were found in the DEX file, where only in the last case the loaded

[14] The difference is that PathClassLoader is unable to load the zipped DEX file.

library and its path was located. In the other two cases the value of the parameter for the path of the .SO file could not be obtained. This could be due to various reasons such as a path parameter for the .SO file being split into different string variables. However, we note that our method still returns the address of each API used to call external files.

Validation and reliability of RAPID

This section briefly describes how validation of RAPID was examined (Section (Validation)) as well as how the reliability of RAPID was tested (Section (Reliability)).

Both tests were conducted on 11,711 APK files where 1260 were malicious samples from the Android Malware Genome Project[15] (Zhou and Jiang, 2012) and 10,451 free applications considered as benign samples downloaded from Google Play. These collected samples cover most of the categories available in the store, i.e., we cover 24/27 of the main/application categories and 17/17 of the games category.[16] All samples were downloaded starting at the end of 2015 and the last update was performed in January, 2016. The popularity of the applications ranged from less than 1000 downloads to prominent applications with millions of downloads like Facebook or YouTube. We decided to use malware and benign samples in our testing as (i) practitioners are usually tasked with malware analysis and thus analyzing Android malware is a highly probable use-case and (ii) we were not sure if malware and benign samples differed significantly, which could lead to potential RAPID errors — our goal was to have diverse Android application coverage in order generalize the validity and reliability of our approach.

Validation

To validate RAPID, we performed cross-comparison to the data in smali files generated by Baksmali, which included three tests for the string object, method object and codeBlock/instruction object. All three yielded the same results verifying the correctness of our approach. The first two tests (string and method component) were implemented by an automatic comparison and was based on 11,705 samples (6 samples could not be decompiled using Baksmali (see Section (Reliability))). The third test was more complex and required manual analysis.

Strings. For this test, we extracted all strings with RAPID as well as from the smali files and ran a cross-comparison. All RAPID strings were found in the smali files and vice versa. Note that the same string may be represented differently in DEX and smali files, e.g., the symbol “"” is represented as ‘\"’ (additional backslash) in smali code as it is a reserved symbol by smali. Our comparison script considered those situations.

Classes and methods. This test focused on method-related data which included the elements that can represent an independent method; method name, class name, type of return value and parameter type. For this purpose our prototype extracted the relevant data from the smali code using regular expressions and utilized our method component. A cross comparison showed that both results coincided.

CodeBlocks and instructions. The last test was rather complex and therefore conducted manually. The problem is that Baksmali includes additional strings/symbols to ease readability which are not part of the original DEX file. To achieve this automatically, it would be necessary to also add these strings which would then correspond to Baksmali code. For instance, the decompiler adds .method to indicate the start of a method. As a result, we could not find any differences between Baksmali and RAPID within the 20 codeBlocks that were tested manually.

Reliability

For this test, we compared the reliability of RAPID again to other prominent approaches — Baksmali and Dex2jar (due to the complexity of the test and the availability of the tools, testing all the tools is outside the scope of our work). The test is successful if the smali code or the JAR file are generated without errors by Baksmali or Dex2jar, respectively. For RAPID, we required that all four parsing levels were executed.

While RAPID successfully parsed all applications, Baksmali as well as Dex2jar failed to process several of them. In detail, Baksmali failed on six applications (error messages were printed and no smali files were generated) and Dex2jar failed on 10 cases to generate a JAR file or the JAR file was corrupt. Surprisingly, all these applications were benign.

The reasons of resulting in such failures varied. In order to be successful, Baksmali and Dex2jar need valid program logic throughout the DEX file. That means, if they parse segments containing errors, exceptions will be thrown and the parser stops (even though the code is never executed). On the contrary, RAPID, as a direct extraction approach, was still able to acquire data from the DEX files on those samples that failed to process.

Experimental results

As discussed in the related work, tools either decompile or convert the binary code and then work on the smali code/JAVA bytecode or implement their own parser to extract the data. Since we cannot compare each individual parser, we only focus on comparing RAPID with smali code and JAVA bytecode (which are the most commonly adopted procedures).

The total runtime T of an approach $A \in \{RAPID, Baksmali, Dex2jar\}$ for m different queries on a single application can be calculated as follows:

$$T^A = T_{unzip} + T^A_{prep} + m \cdot T^A_{query} \tag{1}$$

where T_{unzip} is the time to unzip/decompress the APK file, T_{prep} is the preparation time (decomiling or parsing) and T_{query} is the average time per query.

Since T_{unzip} is independent of the actual approach, we neglect it and separate the efficiency experiment into two

[15] http://www.malgenomeproject.org (last accessed Dec. 6, 2015).

[16] The categories are listed at https://play.google.com/store/apps and then click on ‘Categories’ which is found close the left upper corner of the screen.

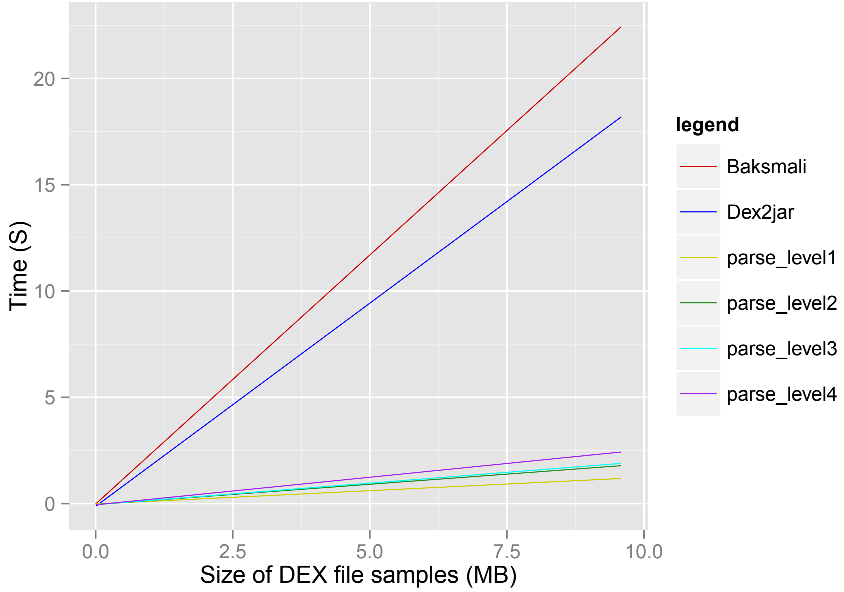

Fig. 3. Regression coefficients for decompiling and parsing for 11,695 applications.

Table 3
Regression coefficients for the different Parsing Levels (PL)s for RAPID as well as the Baksmali and Dex2jar.

	Regression coefficients (s/MB)	Time (min)
Baksmali	2.43	890.27
Dex2jar	1.91	704.34
PL 1	0.13	44.50
PL 2	0.19	64.07
PL 3	0.20	67.73
PL 4	0.26	88.05

sections. First, we analyze T_{prep} which compares the decompiling of the approaches in Section (Decompiling vs parsing). In the subsequent section, T_{query} is analyzed which is the query-time.

The experiments were conducted on an machine with Intel (R) Core (TM) I7-4770S 3.1 GHz CPU, 16 GB memory and Microsoft Windows 7 Professional SP1 64bit.

Decompiling vs parsing

As discussed in Section (Parsing a DEX file), RAPID has different parsing levels and hence the runtime depends on the actual data that is queried. For this test, we measured the runtime for all the four different parsing levels as well as the smali decompilation time and the JAVA bytecode conversion time.

We utilized the sample set introduced in Section (Validation and reliability of RAPID) but excluded the 16 files that couldn't be processed by Dex2jar or Baksmali. Thus, the upcoming tests were conducted on 11,695 samples.

First, we decompressed all the APK files by running a self-implemented JAVA program. Next, we ran Baksmali as well as Dex2jar on the sample set and measured the execution time using a JAVA API. With respect to RAPID, four separate tests were conducted — one per parsing level. Recall, the higher parsing levels include parsing lower levels and thus the time will increase.

The test results are shown in Fig. 3 and clearly demonstrate the performance advantage of RAPID compared to its counterparts. The total runtime and the regression coefficients for each test are provided in Table 3. As shown, the time for Baksmali and Dex2jar are in the same order of magnitude where Dex2jar is insignificantly faster than Baksmali.

Querying data

This second test focused on queries. Note, for this test we only focused on Baksmali and RAPID as parsing the JAVA byte structure is beyond the scope of this paper. However, it is assumed that parsing the JAVA binary files will be similar to parsing the DEX file (both are binary) and thus similar to RAPID.

As will be shown, the query time very much depends on the actual use case and can be very slow for decompiled files. For instance, searching for a single string is less complex than retrieving the class where a method is called or analyzing the invokes from/to a specific function. Therefore, we conducted the test on the previously decompressed 11,695 DEX files. For testing purposes, we devised 3 different use cases/scenarios and measured the time; *string search*, *API search* and *invoke search*.

String search. In the first scenario, we only searched for a specific string. Real world applications of this is if an investigator searches for a specific URL or name. In this scenario we searched for ‘http://’.

For RAPID this meant we only had to construct the string component which is parsing level 1 and run through

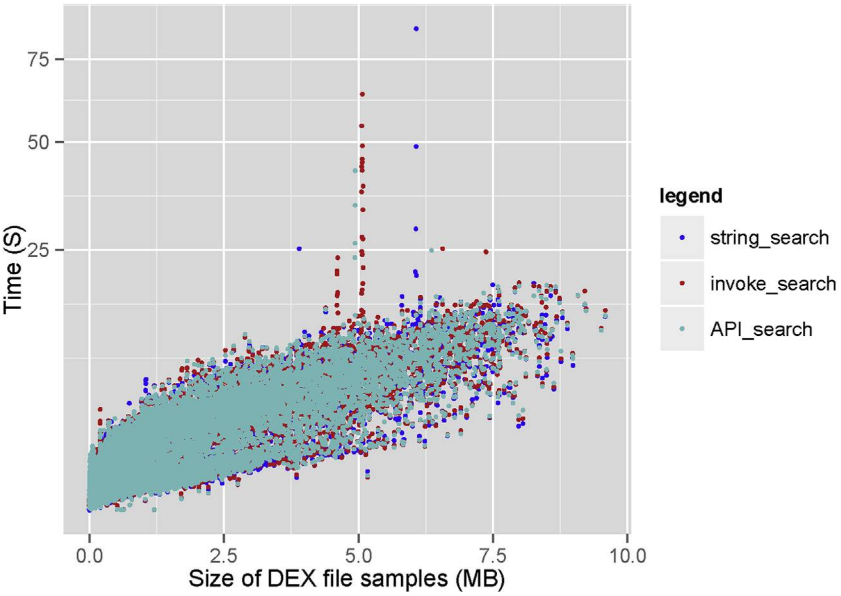

Fig. 4. Times for the smali code searches in *seconds*.

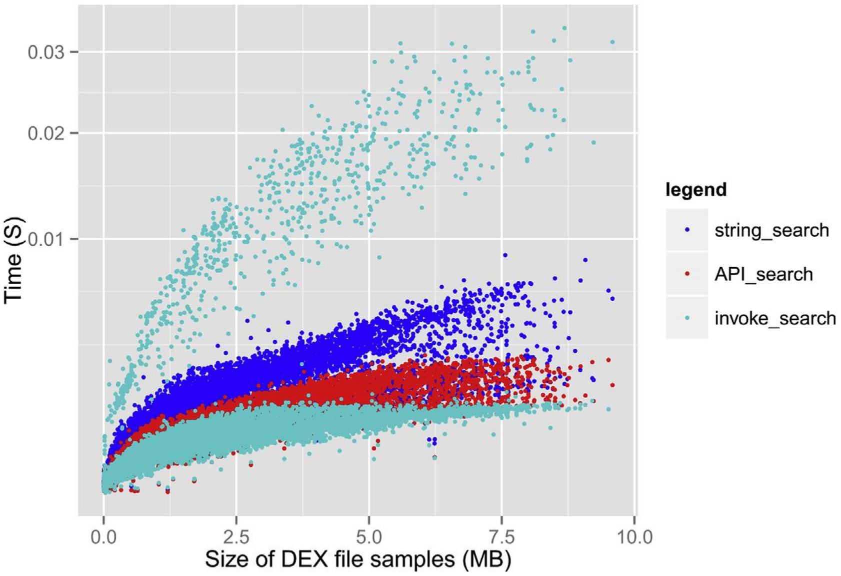

Fig. 5. Times for the RAPID searches in *seconds*.

the linear list. With respect to the smali code, we have to execute a string search on all the decompiled files.

API search. In the second scenario, we looked for the usage of the 'loadLibrary(..)' API in class java.lang.System. For smali code this is similar to a plain string search. Note, although usually more parsing is required (e.g., to analyze the parameters and return value), in this test we focused on finding the API only. With respect to RAPID we can utilize the method component (parsing level 2) to solve this challenge.

Invoke search. The last scenario was the most complex as we aimed at finding the methods that invoke a specific function, i.e., which method/class calls a specific function. In the case of the smali code search, we looked for the function (string search) and then analyzed if this was part of the function and read the class name. RAPID will have to parse all four levels and then start from the instruction object by finding the *opcode* == *invoke*[17]; from there it goes upwards to the codeBlock where a specific instruction object is contained which reveals the *methodId*.

[17] Note, this is simplified for a better understanding, the actual opcode we are searching is invoke-kind.

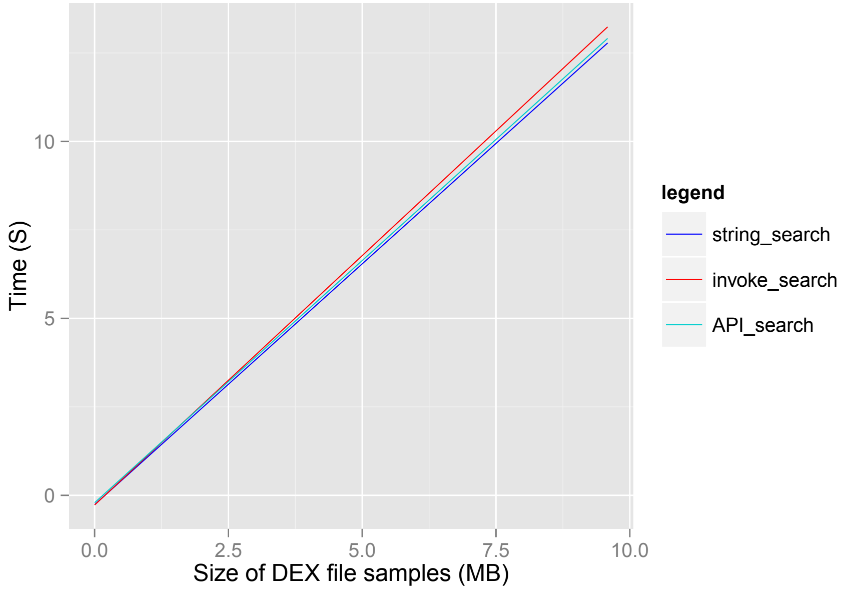

Fig. 6. Linear regression for the smali code searches.

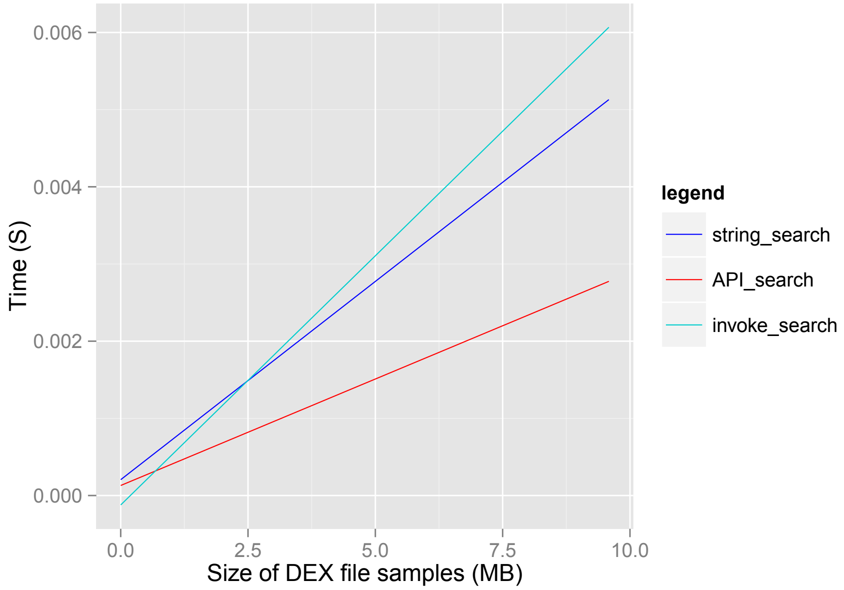

Fig. 7. Linear regression for the RAPID searches.

Results. The results for the different searches (smali queries and RAPID queries) are shown in Figs. 4–7. Note, times are in seconds but the scale is different.

Figs. 4 and 5 show the exact results obtained from both approaches. Focusing on the smali code searches shows that they are similar in time and there are only a few outliers (see Fig. 4). With respect to RAPID, the behavior is quite different. While API/string search behave in a stable manner, there are significant differences for the invoke_search which result from the fact that some APIs are found (slower; points on upper part of the graph) and others are not found. More precisely, in case an API is not found in the application, the algorithm can immediately stop. On the other hand, if the API is found, RAPID then needs to find the invoke which requires more time.

Figs. 6 and 7 show the linear regression obtained from both approaches which could be used to estimate times for different sample sets.

Analyzing the linear regression in more detail reveals the regression coefficients as summarized in Table 4. These coefficients allowed us to upscale the results for larger sample sizes.

Table 4
Regression coefficients for the different searches.

	Regression coefficients (s/MB)	Time
Baksmali		
string search	1.36	467.66 min
API search	1.37	484.46 min
invoke search	1.41	481.34 min
RAPID		
string search	5.138e-04	14.16 s
API search	2.760e-04	7.83 s
invoke search	6.458e-04	13.35 s

Results summary

The previous two sections addressed the processing steps separately. In order to explore the overall performance improvement, this section considers the initial Eq. (1) where we will set $m = 1$ queries, neglect the unzip time $T_{unzip} = 0$ and use the average search times from Table 4. Thus, for the sample set of $n = 11,695$ which equals 22,889.64 MB, the total time for Baksmali is

$$\begin{aligned} T^{Baksmali} &= T_{unzip} + T^{A}_{prep} + m \cdot T^{A}_{query} \\ &= 0 + 890.27\ min + 1 \cdot 477.82\ min \\ &\approx 1,368\ min \end{aligned}$$

and for RAPID it is

$$\begin{aligned} T^{RAPID} &= T_{unzip} + T^{A}_{prep} + m \cdot T^{A}_{query} \\ &= 0 + 88.05\ min + 1 \cdot 0.20\ min \\ &\approx 88\ min \end{aligned}$$

Note that the p-values[18] for all the regression coefficients are significantly less than 0.01.

Limitations

There are four limitations with the current version of RAPID.

First, as shown in Section (Parsing a DEX file), we currently do not parse the complete DEX file and ignore some parts, e.g., sections containing debugging or annotation information (even though they can still be found as scattered strings in the string component). Although our literature review revealed that this data is typically not used, there might be approaches in the future that require this information.

Second, a user needs to get accustomed to the fact that RAPID does not provide a ‘class-object’ as a main component but focuses on strings, methods and codeBlocks.

Users can only retrieve the class data by accessing the class name field of the corresponding method objects.

Third, given the fact that there are currently over 1.5 million applications on market, the test sample size with a little bit over 11,000 files may be considered small.

Finally, although we had some malware samples, we did not experiment with obfuscation and code protection techniques and how RAPID's results might change. For example, it may be possible that current techniques may crash RAPID but pass the validity check of the Android virtual machine.

Conclusion and future work

The problem we tried to solve is that current APK analysis approaches mostly convert the DEX file into intermediate code (e.g., JAVA code, smali code) which is then analyzed or used. This procedure has a significant drawback when it comes to runtime efficiency as one first has to convert everything and then analyze it.

For researchers and practitioners that might have implemented their proprietary DEX parsers for certain Android application analysis work, this means that future work will have to reinvent the wheel since that code is often not being validated and/or shared.

Our idea was to create an easy-to-use library that can be utilized to analyze DEX files. As a result, we presented a new library titled RAPID – a Rapid Android Parser for Investigating DEX files – that directly works on DEX files. In other words, instead of converting the data, we directly extract it.

RAPID is well-documented and comes with multiple APIs that can be utilized by others. As our library is open source and freely available, users can extend it. Our experimental results show the significant performance improvement one can gain using RAPID. Furthermore, we offer the possibility for searching for dynamic loading of libraries which can then support dynamic analysis.

In the future, we will embark on three major steps. First, we want to collect feedback from users regarding the APIs and eventually change or improve the existing set of APIs. Second, we will analyze our code for possible further improvements. Third, we would like to enable RAPID to perform more complex tasks like data-flow or call graph analysis. These can be realized by complex queries which RAPID can handle in a reasonable amount of time.

Appendix A. API summary

Table A.5
RAPID APIs including brief descriptions.

Method	Description
Settings	
setApkDir(String apkDir)	Sets the directory containing the APK/DEX files.
setSingleApk(String apkDir)	To analyze a single APK/DEX file.
setUnzippedFileDir(String unzippedFileDir)	Sets temp-directory for the unzipped DEX files.
getApkList()	Returns a list of all APK files found in the set-directory.
Main object queries (per APK/DEX file)	
getStringList()	Returns a list of string objects parsed out of the current file.
getMethodList()	Returns a list of method objects parsed out of the current file.

[18] p-value can reject the null hypothesis that the slope of the regression line is equal to zero if it is less than the significance levels which researchers always choose 0.01/0.05.

Table A.5 (*continued*)

Method	Description
getCodeBlockList()	Returns a list of codeBlock objects parsed out of the current file.
Search operations	
doesStringExist(String keyword)	Returns true if ‘keyword’ is found in DEX file.
doesMethodExist(MethodElement method)	Returns true if a method exists in DEX file.
doesApiExist(MethodElement api)	Returns true if an API call exists in a DEX file.
getApiList()	Returns a list of all utilized APIs in a DEX file.
getClassList()	Returns a list of class names in DEX file.
getCodeBlockById(int methodId)	Returns the codeBlock of a method according to methodId.
searchStringContaines(String keyword)	Search ‘keyword’ and returns a list of string objects.
searchMethod(MethodElement method)	Returns a list of methodElements (e.g, useful for overloaded methods).
searchInstruction(Instruction[] targetIns)	Return a list of instructions objects.
searchInstruction(String opcode, long operand)	Return a list of instructions objects with same opcode and operand.
searchInsWithOpc(String opcode)	Return a list of instructions objects with same opcode.
searchInsWithString(String stringContent)	Return a list of instructions objects where a string is assigned.
searchInsWithString(StringElement string)	Same than before but search a string object.
Workflow analyses functions	
isMethodInvoked(MethodElement Method)	If a method is invoked/called in DEX file.
areExternalFilesloaded()	Returns true if any external files are loaded.
getMethodInvolker(MethodElement method)	Returns method object list that invokes a specific method.
getExternalFilesDirs()	Returns the directories where the external file(s) are located.
getInsInvokeMethod(MethodElement method)	Return the instruction objects (as list) that invokes a method.
getInsInvokeMethods(MethodElement [] methods)	Same than before but accepts an array of methods.
getInsLoadExternalFiles()	Returns a list of instructions that loads external files.

References

Apvrille A, Nigam R. Obfuscation in android malware, and how to fight back. Virus Bull 2014:1–10.

Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K, Siemens C. Drebin: effective and explainable detection of android malware in your pocket. In: Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS); 2014.

Castillo CA. Android malware past, present, and future. White Paper of McAfee Mobile Security Working Group. 2011. URL: http://www.mcafee.com/us/resources/white-papers/wp-android-malware-past-present-future.pdf.

Chen KZ, Johnson NM, D'Silva V, Dai S, MacNamara K, Magrino TR, et al. Contextual policy enforcement in android applications with permission event graphs. In: NDSS; 2013.

Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services; 2011. p. 239–52. ACM.

Desnos A. Androguard-reverse engineering, malware and goodware analysis of android applications. URL code. google. com/p/androguard 2013.

Desnos A, Gueguen G. Android: from reversing to decompilation. Proc Black Hat Abu Dhabi 2011:77–101. Github link: https://github.com/androguard/androguard.

Dmitrienko A, Liebchen C, Rossow C, Sadeghi A-R. On the (in) security of mobile two-factor authentication. In: Financial Cryptography and Data Security. Springer; 2014. p. 365–83.

Drake JJ, Lanier Z, Mulliner C, Fora PO, Ridley SA, Wicherski G. Android hacker's handbook. John Wiley & Sons; 2014.

Elish KO, Shu X, Yao DD, Ryder BG, Jiang X. Profiling user-trigger dependence for android malware detection. Comput Secur 2015;49:255–73.

Enck W, Octeau D, McDaniel P, Chaudhuri S. A study of android application security. In: USENIX Security Symposium, 2; 2011. p. 2.

Gibler C, Crussell J, Erickson J, Chen H. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. Springer; 2012.

Google. Android dalvik executable format page. Google. 2008. https://source.android.com/devices/tech/dalvik/dex-format.html.

Hex-Rays. IDA Pro. 2005. http://www.hex-rays.com/products/ida/.

Hoffmann J, Ussath M, Holz T, Spreitzenbarth M. Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing; 2013. p. 1844–51. ACM.

Lu L, Li Z, Wu Z, Lee W, Jiang G. Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security; 2012. p. 229–40. ACM.

Peiravian N, Zhu X. Machine learning for android malware detection using permission and API calls. In: Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence ICTAI ’13. Washington, DC, USA: IEEE Computer Society; 2013. p. 300–5.

Seo S-H, Gupta A, Sallam AM, Bertino E, Yim K. Detecting mobile malware threats to homeland security through static analysis. J Netw Comput Appl 2014;38:43–53.

Talha KA, Alper DI, Aydin C. Apk auditor: permission-based android malware detection system. Digit Investig 2015;13:1–14.

Wu D-J, Mao C-H, Wei T-E, Lee H-M, Wu K-P. Droidmat: android malware detection through manifest and API calls tracing. In: Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference. IEEE; 2012. p. 62–9.

Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang XS. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security; 2013. p. 1043–54. ACM.

Zheng C, Zhu S, Dai S, Gu G, Gong X, Han X, et al. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices; 2012. p. 93–104. ACM.

Zheng M, Sun M, Lui J. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference. IEEE; 2013. p. 163–71.

Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on data and Application Security and Privacy CODASPY ’12. New York, NY, USA: ACM; 2012. p. 317–26.

Zhou Y, Jiang X. Dissecting android malware: characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium. IEEE; 2012. p. 95–109.