Android Application Analysis using Reverse Engineering Techniques and Taint-aware Slicing

Syed Arshad
M.Tech CSE
Dept. of Computer Science & Engineering
Mangalore Institute of Technology & Engineering

Ashwin Kumar
Senior Assistant Professor
Dept. of Computer Science & Engineering
Mangalore Institute of Technology & Engineering

ABSTRACT
Android is a victim of its own success, not just in the way it has attracted malicious attention, but in its very nature. One of the reasons the OS has succeeded in gaining market share so rapidly is that it is open source; it is essentially free for manufacturers to implement. Android platform provide only coarse-grained permissions to users with regard to how third party applications use sensitive private data. Malicious applications pose a threat to the security of the Android platform. The growing amount and diversity of these applications render conventional defenses largely ineffective and thus Android smartphones often remain unprotected from novel malware.

In this paper, we propose AT2: "**Android Taint Analysis Tool**", a lightweight tool uses static method for analyzing Android applications (APKs) and generating a detailed report of the analysis performed. AT2 is a tool which performs a static analysis, gathering as many features of an application as possible. AT2 analyzes Smali code, a disassembled version of the DEX format used by Android's Java VM implementation. The provided application is sliced in order to perform data-flow analyses to backtrack parameters used by a given method. This helps to identify suspicious code regions in an automated way. Several other analysis techniques such as visualization of control flow graphs or identification of ad-related code is also possible.

General Terms
Reverse Engineering, Decompilation, Verification, Security

Keywords
Mobile Malware, Taint Analysis, Android, Static Analysis Tool

1. INTRODUCTION
Smartphone malware is on the rise and with 99% of known malware targeting Google's Android platform which is also the most popular mobile platform in the world by a tremendous margin. Users must start making an effort to protect themselves against various threats. The latest example of the terrifying possibilities out there comes from Trustwave security researcher Neal Hindocha [1], who built a proof-of-concept that could be one of the most troubling examples of smartphone malware we've seen to date. Hindocha created code that is capable of tracking a user's taps and swipes as they operate a smartphone. With similar malware, a malicious hacker might be able to steal PINs, account numbers, passwords and other sensitive information user's type into their handsets. Even the touches and swipes recorded over a period of timeis a potential threat to the user.

Android is a main actor in the operating system market for mobile and embedded devices such as mobile phones, tablets and televisions. It is an operating system for such devices, whose upper layers are written in a programming language, also called Android. As a language, Android is Java with an extended library for mobile and interactive applications, hence based on an event-driven architecture. Any Java compiler can compile Android applications, but the resulting Java bytecode must be translated into a final, much optimized, Dalvik bytecode [2] to be run on the device.

As smartphones become more widespread, their users' privacy and security become critical issues. For example, a Wall Street Journal study of iOS and Android applications revealed that 46–55% of smartphone applications transmit users' private information such as location and device ID over networks without users' awareness or consent. Worse, many users are enticed to download and run smartphone applications without carefully understanding the consequences of accepting permissions prompted before installation. This can easily lead to installation of malicious applications.

Sensitive information on smartphones comes from various sources, including sources originating from smartphones themselves and sources received from the Internet. On one hand, smartphones themselves generate sensitive information such as photos, GPS locations, and device identifiers (IMEIs/EIDs). On the other hand, smartphones can receive sensitive information from a plethora of possible sources over the Internet. For example, users may check their bank accounts via a browser or a bank-provided application. Similarly, smartphones are often used for checking email contents from servers such as Gmail or Microsoft Live account. Privacy can be easily invaded if sensitive data from one source were sent to another irrelevant destination.

Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage [3]. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners use their device both for private and work-related communication. Furthermore, the vast majority of users install apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

These problems are well known, and indeed the Android platform does implement state-of-the-practice measures to impede attacks. The Android platform is built as a stack, with various layers running on top of each other. The lower levels consist of an embedded Linux system and its libraries, with Android applications residing at the very top.

Users typically acquire these applications through various channels (e.g., the Google Play Store [4], APKdownloads [5], etc.). The underlying embedded Linux system provides the enforcement mechanisms common to the Linux kernel, such as a user-based permission model, process isolation and

secure inter-process communication. By default, an application is not allowed to directly interact with other applications, operating system processes, or a user's private data. The latter includes, for example, access to the contacts list. Android regulates access to such private data via a permission-based security model where, to access security-sensitive API functions, applications have to statically declare the permissions they require. An application may only be installed following the user's consent, yet users currently have little control over the installation process, as they must either grant all of the permissions that an app demands, or else forego installation. The problem is aggravated by the coarse-grained nature of Android permissions [6]. Android's existing permission system does not allow fine-grained restrictions on information flow, as a result of this limitation, users grant too many permissions too often, thus running the risk to give malicious apps access to private data.

2. BACKGROUND AND EXAMPLE

This section shows and example code and reviews the concepts behind Program slicing and Static taint analysis. It then introduces the AT2 tool for analyzing android packages, including details of its implementation.

The example code shown in the below reads a password from a text field (line5) whenever the application is restarted. When the user clicks on a button of the activity, it is sent to some constant telephone number via SMS (line22). This constitutes a data flow from the password field (the source) to the SMS API (the sink). Though this is a small example, similar code is known to exist in real-world malware apps [7].

```
1  public class LeakageApp extends Activity{
2    private User user = null;
3    protected void onRestart(){
4      EditText usernameText =
         (EditText)findViewById(R.id.username);
5      EditText passwordText =
         (EditText)findViewById(R.id.password);
6      String uname = usernameText.toString();
7      String pwd = passwordText.toString();
8      this.user = new User(uname, pwd);
9    }
10   //Callback method; name defined in Layout-XML
11   public void sendMessage(View view){
12     if(user != null){
13       Password pwdObject = user.getPwdObject();
14       String password = pwdObject.getPassword();
15       String obfPwd = ""; //must track primitives
16       for(char c : password.toCharArray())
17         obfPwd += c + "_"; //must handle concat.
18
19       String message = "User: " +
20          user.getUsername() + " | Pwd: " + obfPwd;
21       SmsManager sms = SmsManager.getDefault();
22       sms.sendTextMessage("+44 020 7321 0905", null,
23         message, null, null);
24     }
25   }
26 }
```

In this above example, sendMessage() is associated with a button in the app's UI. It is a callback method that gets triggered by an onClickevent. In Android, listeners are defined either directly in the code or in the layout XML file, as is assumed here. Thus, analyzing the source code alone is insufficient—one must also process the meta data files to correctly associate all callback methods.

In this code a leak only occurs if onRestart() is called, initializing the user variable, before sendMessage() executes.

AT2 uses Static android analysis framework which covers an important aspect of an app analysis process: automated static analysis. Here implemented variant of data-flow analysis [8, 9], namely program slicing [10], enables the proposed tool to automatically search for constant values which are used as parameters in defined method invocations. This way, the analyst can for example determine if an application is able to send short messages to a hardcoded number—which would result in a strong misuse potential of this application. This search is called static backtracking. Based on these results the analyst can, e.g., let some heuristic decide which apps are worth a more thorough inspection because they might exhibit malicious behavior: long sleep intervals, hardcoded telephone numbers, calls to sudo and so on.

Using AT2 analyst can also perform a manual inspection. After an application is loaded within the tool, the analyst has access to options such as:

- Navigate through the application contents which are presented in a tree structure. Smali and optionally decompiled Java code is accessible, which is colored, and links to labels and methods are clickable.
- Control flow graphs (CFGs) can be generated and exported.
- AT2 offers the possibility to search for several program components, e. g., strings and invocations.
- AT2 knows about ad package paths and can ignore classes inside them.

An automatic static analysis should run in the background, possibly on a large set of applications. AT2 offers a lot of command line options to properly work without a GUI.

Being a static analyzer, AT2 is expected to work fast for our use case. Many applications need to be analyzed in a short amount of time to quickly get an idea which applications need to be investigated more closely by means of a more expensive manual or dynamic analysis. A static analysis of a typical app from our evaluation set is completed in less than 10 seconds on average. Sometimes the process is even faster, if the application is small.

3. STATIC BACKTRACKING

The ability to perform static data-flow analyses of method parameters (called static backtracking in this paper) is one of the core components of AT2. It enables the analyst to define a set of methods of interest with their respective signature (parameters), in order to see whether they obtain any constants as inputfor example, the analyst wants to determine if some application is able to send short messages to a hardcoded number or with any hardcoded message text—both of which indicate a suspicious usage of this feature.

3.1 General Workflow

AT2 is based on static analysis methods and thus the first step is to dissect Android applications. Such applications are packaged in APK files, which are more or less ZIP compressed files with the compiled bytecode, additional metadata such as the Manifest file, and additional resources such as image or audio files.

AT2 unpacks these APK files in the following way in order to perform the data-flow analysis and further analysis operations:

- The analyst loads an Android application (APK file) or specifies at least one from the command line.

- AT2 unpacks the contents of the app and generates smali files for all classes, using the android-apktool. Working directly on the bytecode enables us to obtained a detailed view of the code and overcomes limitations of tools that rely on decompiling the bytecode to Java code [11].

- Then parses the smali files and creates an appropriate object representation of its contents. At this point, the static analysis can begin since all relevant information is unpacked and available in a usable form for further processing.

AT2 will then perform the program slicing [10], which is explained in the next section. The complete process of AT2 is shown in Fig 1 below.

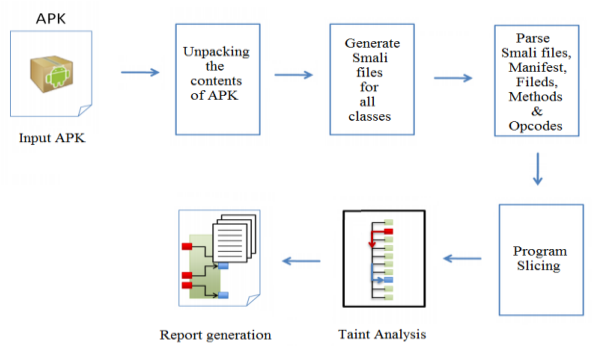

Fig.1 Process of AT2

3.2 Program Slicing

In order to perform backtracking of method parameters and to perform the slicing, a slicing criterion must be defined. In our case, the criterion consists of the following information: method name and full class name of its corresponding class, method signature, and the index of the parameter that shall be backtracked. The slicing criterion fully specifies the relevant opcodes that invoke the desired methods in the analyzed application. Such a criterion enables to search for use-def chains. The first search will be through all invoke opcodes for matching ones. Afterwards, the given parameter index is translated to a particularly used register in the decompiled code (use information). The previous opcodes in the corresponding basic blocks are checked and determined whether the opcodes perform some operation with the currently tracked register. In other words, backward slicing is performed. Generally speaking, all opcodes that modify or use the tracked register will be checked and will backtrack all the interactions until constant (def information) is found.

AT2 has an internal queue where all registers are stored which have not yet been backtracked. The queue is initially filled with the registers found during the first search for matching invokes opcodes and AT2 backtracks each register until the queue is empty.

It is eventually filled as the logic finds opcodes inferring with the tracked register that make use of additional registers. The queue stores the registers name and its exact opcode location in the program in order to backtrack it at some time later. If a tracked register vx is overwritten by register vy by the means of a move opcode, register vy will of course be backtracked from this instruction on instead of vx; this is called aliasing [12]. The same is true for all opcodes that put a result into the tracked register: all involved registers are added to the queue and are later backtracked. If the tracked register itself is not part of the value registers, it will not be backtracked anymore. Until a found constant terminates the backward slicing, several opcodes require special handling in order to find constants of interest. Due to space constraints, it is hard to describe in detail how the proposed tool, AT2 deals with arrays, fields, basic block boundaries, method invocations, return values and the like. However, AT2 handles all opcodes and employs a combination of backward and forward slicing to find all constants which might get assigned as a parameter to the slicing criterion.

3.3 Constants

The analysis process is terminated when one of the following conditions holds:

- A constant value is assigned to the tracked register.

- An object reference is written into the tracked register.

These two cases end the search for constants for a tracked register: the first marks our goal to find assigned constants in the bytecode which finishes our search for def information. All opcodes of the const-x type provide such information in addition to some others, e. g., mathematical operations or initialized fields and arrays. They assign constants to registers, e. g., strings or integers. In both cases the register will beoverwritten and has an unknown semantically meaning before the assignment, which is irrelevant for our analysis. While the first one adds a resulting constant to our search, the second one terminates our search. If the register is overwritten with some reference, still all involved constants for this object can be seen.

Apart from opcodes that put a constant of a specific type into the tracked register, the following aspects may be encountered during the search, if they are somehow linked to the tracked register r:

- Fields and arrays with their types, names, initial and assigned values if a value is copied from them to r.

- Unknown (API) methods if they are called and return a value which is assigned to r. Known methods are part on the usedef chain and all return values are tracked.

- Variable names and types for found constants.

- Opcodes that overwrite r with something else, e. g., if an exception is moved to it.

If such cases are found, they are added to the result set in a proper format and are tagged accordingly. These results store additional meta-information such as the line number, the filename, and other relevant information that is helpful during the analysis.

3.4 Static Analysis

Static analyses inspect the program code to derive information about the program's behavior at runtime. As nearly every program has variable ingredients (inputs from a user, files, the internet etc.) an analysis has to abstract from concrete program runs. Instead it aims to cover all possibilities by making conservative assumptions. The properties derived from these assumptions can be weaker than the program's properties actually are, but they are guaranteed to be applicable for every program run. In this way the analysis detects a program behavior which might not actually happen during runtime, but it does not miss a behavior which can happen during runtime (i.e. leakage of sensitive data). If an analysis features this over approximation, it is sound.

Static analysis has many fields of application. Besides checking for programming errors and security flaws, which aim at the correctness of a program, there are many analyses included in modern compilers which try to optimize programs.

4. IMPLEMENTATION

AT2 tool is implemented in Java as well as Python on Linux Platform.HTML, CSS and JavaScript has been used to develop the tool's reporting process. The program slicing is performed on method name and full class name. The slicing criterion fully specifies the relevant opcodes that invoke the desired methods in the analyzed application. The required Ad Networks, backtracking patterns, heuristic patterns and permissions are defined in the appropriate xml files and xml-schema files used to validate the xml files. Soot: Java optimization framework is used to perform static taint analysis on the sliced programs and generate CFGs of the required methods and also to generate the analysis report in HTML/Text format.

5. CONCLUSION

In this paper AT2 tool is introduced, which analyses the android applications statically. AT2 performs data-flow analysis based on program slicing to analyze the structure of an application and also tainting is performed to detect the information leakages. Using this tool the users or analysts can analyze the android packages (APK's) for security issues. The complete user friendly analysis reports are presented to the user.

6. REFERENCES

[1] Researcher to demo hack for logging Android, iOS touchscreen movements - January 30, 2014

[2] http://www.scmagazine.com/researcher-to-demo-hack-for-logging-android-ios-touchscreen-movements/article/331894/

[3] Bytecode for the Dalvik VM, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

[4] Your Apps Are Watching You - http://online.wsj.com/news/articles/SB10001424052748704694004576020083703574602

[5] Google Play - https://play.google.com/store?hl=en

[6] APKdownloads - http://www.apkdownloads.com

[7] The Effectiveness of Application Permissions - Usenix - www.usenix.org/event/webapps11/tech/final_files/Felt.pdf

[8] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[9] F. E. Allen and J. Cocke. A program data flow analysis procedure. Commun. ACM, 19(3), Mar. 1976.

[10] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. ACM Comput. Surv., 8(3), Sept. 1976.

[11] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. SIGPLAN Not., 25(6), June 1990.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In USENIX Security Symposium, 2011

[13] G. Ramalingam. The undecidability of aliasing. ACM Trans. Program. Lang. Syst., 16(5), Sept. 1994.

[14] Soot: a Java Optimization Framework - http://www.sable.mcgill.ca/soot/

[15] Highly Precise Taint Analysis for Android Applications 2013 - Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel - Secure Software Engineering Group, EC SPRIDE,SnT, University of Luxembourg - Penn State University

[16] All You Ever Wanted to Know About - Dynamic Taint Analysis and Forward Symbolic Execution(but might have been afraid to ask) 2009 - Edward J. Schwartz, Thanassis Avgerinos, David Brumley - Carnegie Mellon University Pittsburgh, PA

[17] Moutaz Alazab, Veelasha Monsamy, Lynn Batten, Patrik Lantz, andRonghua Tian. Analysis of malicious and benign android applications.In Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on, pages 608–616. IEEE, 2012.

[18] Glenn Ammons, Rastislav Bod´ık, and James R Larus. Mining specifications. In ACM Sigplan Notices, volume 37, pages 4–16. ACM, 2002.

[19] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, andAhmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigateprivilege escalation attacks. Technische Universit¨at Darmstadt, TechnicalReport TR-2011-04, 2011.

[20] Patrick PF Chan, Lucas CK Hui, and SM Yiu. Droidchecker: analyzingandroid applications for capability leak. In Proceedings of the fifth ACMconference on Security and Privacy in Wireless and Mobile Networks,pages 125–136. ACM, 2012.