September 16, 2019

Finding (partial) code clones at method level in Android programs without access to source code to detect copyright infringements or security issues

Armijn Hemel

Finding (partial) code clones at method level in Android programs without access to source code to detect copyright infringements or security issues

Abstract

Nearly all programs for Android devices are distributed without source code being made available. This means that it is a lot harder to do audits of these programs for for example copyright infringement detection or security issue detection. By examining individual methods inside an Android program and comparing these to a database of methods from known programs it is possible to make an educated guess of which programs or program fragments are used in the program, and possibly detect copyright infringements or trojaned versions of programs.

Keywords

android, dalvik, dex, code clones, code clone detection, binary, apps, app store, Linux, copyright infringement, security, malware, malware detection, locality sensitive hashing

Background

Nearly all programs for Android devices are distributed without source code being made available. Programs for Android are typically installed through a variety of app stores (Google Play Store, F-Droid, as well as vendor specific app stores). Users search for an application in the app store, or get redirected to the right entry in the app store via a link and then install the application by clicking on a button saying “Install now” or similar, after which the app is installed.

These app stores contain hundreds of thousands of programs, with new programs or new versions of programs being added daily. The sheer volume and the absence of source code makes it a challenge to analyze the packages for copyright infringement and security issues. While the former is largely a legal problem typically not affecting users the latter is a big problem, possibly leading to identity theft and other types of damage.

Copyright infringement

Many of the programs in the Android app store are built on open source software. Examples are games that use open source gaming engines, or applications using compression libraries or graphics libraries. Depending on the license of the software used different rules have to be followed, such as disclosure of source code under the same or similar license, or attribution for authors. These conditions are very often not followed, leading to copyright infringement and copyright infringement lawsuits. Detecting possible open source license violation without access to source code is more difficult than when access to source code is available[1].

Security issues

Security in apps is a big concern, as it can possibly lead to disclosure of sensitive information, identity theft and other types of damage[3]. Without having access to source code finding security problems are harder and sometimes near impossible to detect.
One problem is bugs, due to sloppy programming or using outdated (third party) components.

Another problem is malware disguised as regular apps. In the past there have been problems with programs being repackaged as regular programs and being added to app stores[3][4], or even being preinstalled on devices[5][6]. These files are usually largely unmodified to be able to pass as the original program, but some code has been replaced by malware, or extra code with malware has been inserted.

Proposed method

The core of the problem is finding out where software originally came from, ideally going back to source code, but this is not always possible. Being able to (partially) match programs to known binaries (with or without source code) is already a step forward and allows us to make an educated guess about the program.

Android package structure

An Android package is a ZIP archive with in it several files, such as:

- resource files, such as graphics and media files
- meta information related to packaging
- external third party libraries used by the main application
- an application, possibly divided across multiple files, but usually contained in a file called “classes.dex”, although the byte code might also be in different files (for example in the OAT file format)

This disclosure focuses on the application and files in the Dalvik file format (the “.dex” files). The Dalvik format[7] mainly consists of compiled code, with lookup tables with extra information, such as type information, strings, and so on. There are various versions of the Dalvik format in circulation, including “optimized DEX” (odex) and OAT.

The code inside Dalvik files is very structured: each method in source code directly maps to a so called “code item” in the Dalvik binary file. A code item consists of some metadata and an array of byte code. This byte code consists of instructions with “op codes” and parameters[8][9]. How the byte code is generated depends on the build environment that is used (version of the Android compiler that is used, the optimizations chosen, and so on).

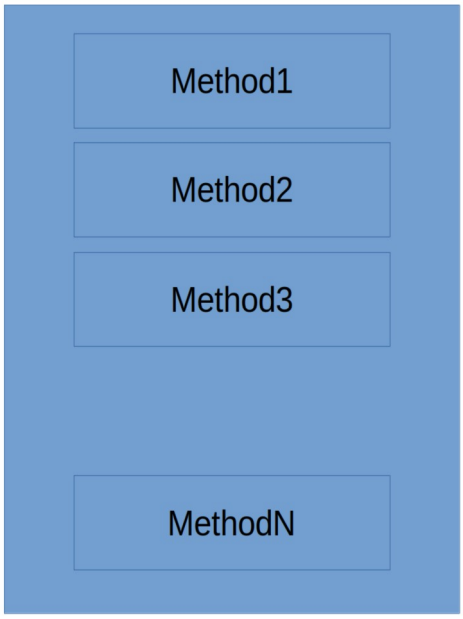

Figure 1: Structure of Dalvik Dex byte code

These blocks of byte code can be carved from the Dalvik executable. This makes it possible to divide each Dalvik file into chunks, with each chunk from the binary corresponding to an individual method inside the Dalvik source code.

Cryptographic hashing

For each of these method chunks with instructions a checksum, such as MD5, SHA1, SHA256, SHA512 or similar, can be calculated. These hashes can be stored in a database, together with meta information about the Dalvik file they were carved from, such as, but not limited to:

- file name
- download location
- app store name
- meta information about the publisher
- build environment (if known)
- known security issues

Hashes obtained for methods can be compared to hashes stored in the database to see if there is an identical match. In case a match is found then the code for the method is identical to a method in another binary and likely also built in the same or near same build environment.

In case the source code for the match found in the database is known, then it automatically means that the code for the method is also known and can be analyzed for security defects, copyright issues, and so on.

Example: a method “methodA” from package A, for which no source code is available, matches with “methodB” from package B, for which source code is available. The source code for “methodA” will then also be known (because it will be identical to the source code for “methodB”).

Another application for this method would be to compare the methods of a (suspected) trojaned version of a program to the methods of the original program to find out which parts have been modified. The assumption is that trojaned versions of programs will try to stay as close to the original as possible, and only insert or replace code at a few places. These places can be found by looking for methods in the Dex bytecode that are different from the known program, instead of identical.

Example: a method “showCredits” from package “GameA” suspected to be a trojan is not the same as the method “showCredits” from the official package “GameA”. This could indicate that the malware is hidden in this method, and it would allow malware researchers to zoom in on this particular method and prioritize their efforts, and ignore all the other known methods.

Figure 2: Comparing individual methods in two different Dalvik Dex bytecode files

Locality sensitive hashing

If code has been slightly modified then the method using hashes such as described above will not always find matches: a difference of even a single bit will lead to radically different hash results. While this is useful when comparing programs that are already known (such as the suspected trojan example) it won't be useful in case a single operand to an op code has been changed (for example, changing an integer supplied to a method call).

In that case it would be useful to use locality sensitive hashing (LSH) instead of regular cryptographic hashes. When comparing LSH hashes a distance is computed, which says something about the similarity of the data the hashes were computed for. Well known hashes are ssdeep[10] and TLSH[11]. Using locality sensitive matching would make it possible to say something about how big the size of the change is.

Matching code structure only

As a refinement the methods could first be stripped of parameters to the op codes so the comparison can be done on the code structure only. As soon as parameters for the op codes (for example: arguments to method calls, assignment of variables, and so on) vary the cryptographic hashes of the methods will change. By first removing the parameters to the op codes from the byte code and only focusing on the op codes matches can be done on just the code structure. While a match for code structure only will not be as accurate as a match for a full method it will reveal some information about the code.

Related work and innovation

This disclosure is not the first publication in this field. There have been other publications. Most notably are publications about AnDarwin[12], DroidMOSS[13] and APKLancet[14]. All three approaches use decompilers to extract op codes from the original byte code.

The innovation in this disclosure is that no decompilation is done: the byte code is used “as is” without decompilation. No extra knowledge about structure of the file, or known malware, is fed into the matching system. Another benefit of this system is that comparing hashes without decompilation is very quick and highly parallelizable, as most of the inputs (namely hashes of methods in other files) can all be computed beforehand and only need to be stored in for example a database.

A drawback is that it is possibly less accurate than methods described in the other publications.

Claims

This invention claims the following:

1. a system that receives an Android bytecode file and splits the bytecode into individual methods, and that computes a cryptographic hash (such as MD5, SHA1, SHA256, etc.) of the contents of each method
2. the method of claim 1 wherein the hashes are stored into a database together with meta information about the Android bytecode file which could include the file name, download location, app store location, meta information about the publisher (name, location), build environment, and so on
3. the method of claim 2 wherein hash values of one or methods are looked up in a database containing hashes of methods
4. the method of claim 3 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported
5. a system that receives an Android bytecode file and splits the bytecode into individual methods and that computes a locality sensitive hash (such as TLSH, ssdeep, etc.) of the contents of each method
6. the method of claim 5 wherein the locality sensitive hash values are stored into a database together with meta information about the Android bytecode file which could include the file name, download location, app store location, meta information about the publisher (name, location), build environment, and so on
7. the method of claim 6 wherein a locality sensitive hash of a method is compared with hashes stored in the database and the methods that have the closest distance are determined and reported

8. the method of claim 7 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported
9. the method of claim 6 wherein additionally a cryptographic hash (such as MD5, SHA1, SHA256, etc.) for a method is stored
10. the method of claim 9 wherein for an Android bytecode file for each method a cryptographic hash is computed; the hash is looked up in the database and for each match the file in which the match was found is stored. For methods for which there are no matches found a locality sensitive hash is computed and compared to hashes of methods in the database, but only limited to methods occuring in files for which there already were matches. The methods that have the closest distances are reported.
11. the method of claim 10 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported
12. a system that receives an Android bytecode file and splits the bytecode into individual methods, extracts op codes, discarding the parameters and that computes a cryptographic hash (such as MD5, SHA1, SHA256, etc.) of the concatenation of the op codes in the exact same order as found in the method
13. the method of claim 12 wherein the hashes are stored into a database together with meta information about the Android bytecode file which could include the file name, download location, app store location, meta information about the publisher (name, location), build environment, and so on
14. the method of claim 13 wherein up the hash values of one or more concatenations of op codes of a method is looked up in a database containing hashes of concatenations of op codes of a method
15. the method of claim 14 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported
16. a system that receives an Android bytecode file and splits the bytecode into individual methods, extracts op codes, discarding the parameters and that computes a locality sensitive hash (such as TLSH, ssdeep, etc.) of the concatenation of the op codes in the exact same order as found in the method
17. the method of claim 16 wherein the locality sensitive hash values are stored into a database together with meta information about the Android bytecode file which could include the file name, download location, app store location, meta information about the publisher (name, location), build environment, and so on
18. the method of claim 17 wheren a locality sensitive hash of a concatenation of op codes of a method is compared with hashes stored in the database and the methods that have the closest distance are determined
19. the method of claim 18 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported
20. the method of claim 17 wherein additionally a cryptographic hash (such as MD5, SHA1, SHA256, etc.) for a method is stored
21. the method of claim 20 wherein for an Android bytecode file for each concatenation of op codes of a method a cryptographic hash is computed; the hash is looked up in the database and for each match the file in which the match was found is stored. For concatenations of op codes of a method for which there are no matches found a locality sensitive hash is computed and compared to hashes of concatenations of op codes methods in the database, but only limited to methods occuring in files for which there already were matches. The methods that have the closest distances are reported.
22. the method of claim 21 wherein any meta data information stored in the database that corresponds to a hash that was matched is reported

References

[1] Finding Software License Violations Through Binary Code Clone Detection, IP.com Disclosure Number: IPCOM000214472D, https://priorart.ip.com/IPCOM/000214472

[2] Rastogi, Sajal & Bhushan, Kriti & Gupta, B B. (2016). Android Applications Repackaging Detection Techniques for Smartphone Devices. Procedia Computer Science. 78. 26-32. 10.1016/j.procs.2016.02.006.

[3] “Android security: Password-stealing malware sneaks in Google Play store in bogus apps”, http://web.archive.org/web/20190116192326/https://www.zdnet.com/article/android-security-password-stealing-trojan-malware-sneaks-in-google-play-store-in-bogus-apps/

[4] “Can Google win its battle with Android malware?”. http://web.archive.org/web/20190116192339/https://www.zdnet.com/article/can-google-win-its-battle-with-android-malware/

[5] “Pre-Installed Malware Found On 5 Million Popular Android Phones”, http://web.archive.org/web/20190116192752/https://thehackernews.com/2018/03/android-botnet-malware.html

[6] “Android devices ship with pre-installed malware”, http://web.archive.org/web/20190116192804/https://blog.avast.com/android-devices-ship-with-pre-installed-malware

[7] Dalvik Executable format, https://source.android.com/devices/tech/dalvik/dex-format

[8] Dalvik bytecode, https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[9] Dalvik Executable instruction formats, https://source.android.com/devices/tech/dalvik/instruction-formats

[10] ssdeep, https://ssdeep-project.github.io/ssdeep/index.html

[11] TLSH, https://github.com/trendmicro/tlsh

[12] Crussell, Jonathan & Gibler, Clint & Chen, Hao. (2013). AnDarwin: Scalable Detection of Semantically Similar Android Applications. 8134. 182-199. 10.1007/978-3-642-40203-6_11.

[13] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. 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). ACM, New York, NY, USA, 317-326. DOI=http://dx.doi.org/10.1145/2133601.2133640

[14] Yang, Wenbo & Li, Juanru & Zhang, Yuanyuan & Li, Yong & Shu, Junliang & Gu, Dawu. (2014). APKLancet: tumor payload diagnosis and purification for android applications. 10.1145/2590296.2590314.