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

Armijn Hemel

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

Abstract

Many Java programs are distributed in binary form 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 class files inside a Java program and comparing these to a database of class files 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

java, code clones, code clone detection, maven, binary, copyright infringement, security, malware, malware detection, tlsh, locality sensitive hashing

Background

Many Java programs are distributed without source code being made available or not easily discoverable. The most popular way to distribute Java JAR files is through the Maven build tool[2] and the Maven Central Repository[3].

This repository contains millions of programs (often components that are reused), with new programs or new versions of programs being added daily. Sometimes the source code is also available directly from the Maven Central Repository, but due to the code often being released under a license that doesn't require the source code to be released as well, this very often is not the case and the source code is only available from a different site, or not at all.

The sheer volume and the frequent 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/components in the Maven Central Repository 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 programs is a big concern, as it can possibly lead to disclosure of sensitive information, identity theft and other types of damage. 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 programs, or trojans.

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.

Java JAR file package structure

A Java JAR file is a ZIP archive with in it several files, such as:

- Java class files
- dependencies in the form of other JAR files
- meta information related to packaging
- external third party libraries used by the application

This disclosure focuses on the application and files in the Java class files code, with lookup tables with extra information, such as type information, strings, and so on.

Cryptographic hashing

For each of these class files 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 Java JAR file they were extracted from, such as, but not limited to:

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

Hashes obtained for class files 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 class file is identical to a class file in another JAR file 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 class file is also known and can be analyzed for security defects, copyright issues, and so on.

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

Another application for this method would be to compare the class files of a (suspected) trojaned version of a program to the class files 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 Java class files that are different from the known program, instead of identical.

Example: a class file “showCredits.class” from package “GameA” which is suspected to be a trojan (and is called the same as the original version to let people think it is the official version) is not the same as the class file “showCredits.class” from the official package called “GameA”. This could indicate that the malware is hidden in this class file, and it would allow malware researchers to zoom in on this particular class files and prioritize their efforts, and ignore all the other known class 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[4] and TLSH[5]. Using locality sensitive matching would make it possible to say something about how big the size of the change is.

Claims

This invention claims the following:

1. a system that receives a Java class file and computes a cryptographic hash (such as MD5, SHA1, SHA256, etc.) of the contents of said Java class file
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 class files are looked up in a database containing hashes of class files
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 a Java class file and computes a locality sensitive hash (such as TLSH, ssdeep, etc.) of the contents of said Java class file
6. the method of claim 5 wherein the locality sensitive hash values are stored into a database together with meta information about the Java class file which could include the file name, the

JAR archive it was extracted from, download location of the archive, 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 Java class files is compared with hashes stored in the database and the Java class files 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 Java class file is stored
10. the method of claim 9 wherein for aJava class file 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 Java class files for which there are no matches found a locality sensitive hash is computed and compared to hashes of Java class files in the database, but only limited to Java class fiels occuring in JAR files that contain Java class files for which there already were matches. The Java class files 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

References

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

[2] https://en.wikipedia.org/wiki/Apache_Maven

[3] https://mvnrepository.com/repos/central

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

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