Studying the Impact of Obfuscation on Source Code Plagiarism Detection

Deliverable 1: Scope definition

Albert Cabré Juan
20th July 2013

Index

1. Project Formulation

Software plagiarism is the act of using someone else's source code, in whole or in part, into one's own program without respecting the original source code licence. Considering that giant companies have filled billion-dollar lawsuits (Oracle vs Google case), software plagiarism is no longer just a problem in Higher Education but it has become a serious threat to Intellectual Property Protection and to the open source movement, which has been widely adopted by the Software Industry.

In the past, most companies kept their source code as an industrial secret. This was done to prevent plagiarism and because they thought that the least people knew about their code the most difficult it would be for hackers to crack it (Security through obscurity). This trend has quickly changed recently in favor of open source: by letting everyone examine their source code and submit bug reports, the code security is greatly enhanced. However, this approach opens the door for code plagiarism. By making source code public, everyone can copy it.

For the reasons above, great efforts are being done to develop effective plagiarism detection tools. However, current source code plagiarism detection tools are of limited use if the plagiarism is actively disguised through obfuscation. Code obfuscation is the act of modifying a source code in a manner such that the modified version doesn't look like the original code, while preserving the code behavior. While obfuscation can have a lot of legitimate purposes, plagiarists have realized that by obfuscating source code, plagiarism is much harder to detect because the copied code doesn't look the same.

This project will research how code obfuscation impacts current detection approaches by doing a large-scale evaluation of clone and plagiarism detectors.

2. Scope

To study the impact of obfuscation on source code plagiarism detection, an automatic code obfuscation tool will be developed that will apply a wide-range set of obfuscation techniques (by themselves and with different combinations) to source code samples.

Second, a set of plagiarism and code detectors will be fed with the original source code samples and their respective obfuscated versions while recording the different results.

Then, by using data mining techniques, I will study how likely different plagiarism and clone detectors can still detect the obfuscated copy as plagiarized and which obfuscation techniques were the most effective.

Ultimately, I will try to use this information to improve the obfuscation tool and produce a code obfuscator capable of bypassing plagiarism detection tools.

3. Possible obstacles and solutions

3.1 Experiment design

A bad experiment design could easily produce flawed results. Selecting poor code plagiarism detectors, non-representative code samples, choosing non-state-of-the-art obfuscation techniques or failing to properly apply them could all invalidate the study results. The main strategy to overcome this problem is a close supervision of the experiment by my project tutor, who is a clone detection specialist and will be able to properly advise and supervise the experiment.

3.2 Timetable

Because of the special conditions in which this project will be developed (4 month Erasmus), the timetable is a critical part of the project. To overcome it, we'll set a very rigid and realistic timetable with weekly meetings with the project tutor to ensure the project is following the timetable.

3.3 Bugs

Considering that a source code obfuscator is relatively complex software, it's easy to introduce bugs while developing it. Furthermore, because a source code obfuscator output is source code, the bugs could introduce bugs into the code their produce. To ensure no bugs are present, a robust set of automated tests will be run against each output of the source code obfuscator, to ensure that the produced code behaves exactly as the original source code.

3.4 Results interpretation

When dealing with data mining and statistics, being able to properly understand the output of the statistical tests is not a trivial task. Fallacies and subjective bias can play a very significant role when interpreting statistics. To ensure this project's statistical results are properly interpreted, I will use the techniques and methodologies studied in the Data Mining course I took this last semester, ensuring that there's no margin for error.

3.5 Computational power

Some of the statistical tests are quite expensive computationally speaking. Moreover, some of their costs grow very fast as their input grows. Because the number of possible combinations between different obfuscation techniques also grows very fast as the number of techniques grows, there could be a real computational problem if enough obfuscation techniques are evaluated. Shall this be the case there are two solutions:

- First, if the order of magnitude of the problem were big but not really big, usage of UCL hardware would probably be enough.
- If, in the contrary, the order of magnitude is high enough, there's no realistic way to carry the computation on. In this case, the number of inputs must be reduced.

4. Methodology

Because of the tight timetable the project will be developed in, an agile approach seems the best fit. However, currently accepted agile methodologies (Extreme programming, SCRUM...) are very team-oriented, and so strictly following any of them would make no real sense. However, some of these methodologies concepts are still valid for solo projects:

4.1 TDD

Test-driven development is specifically very well suited for projects where bugs need to be diagnosed very early because later fixing suppose a not acceptable cost. In this case, if the obfuscator was to produce bugged code or code that is not behaviorally equivalent to the original and this wasn't detected until a later stage, the whole project could be compromised.

4.2 Short development cycle

By using an iterative approach with short cycles (goals will be specified in a weekly basis), it’s much easier to keep the project on schedule and be conscious about the project current state.

4.3 Intensive client feedback

While the project doesn’t have a real client, this concept from agile methodologies still applies: By letting the person that is ultimately responsible for evaluating the project check the project state as frequently as possible, chances for misunderstandings are greatly reduced and, when they happen, are corrected much faster.

5. Development Tools

Git and Github are going to be the only tools used. Most of the software development tools and services are enormously bloated with features that are very rarely used, especially in the case of a solo project. Git enforces a short cycle approach (by the means of commits) and forces the developer to document the changes he does. Github provides a Git repository and a ticket system, perfect for setting milestones and tracking the progress.

6. Validation methods

The combination of TDD, Github tickets and weekly meetings with the project tutor ensure the validation of the objectives on their own without need for further measures.