Excel@FIT 2015
http://excel.fit.vutbr.cz

Plagiarism Recognizer in PHP Source Code

Ondřej Krpec*

Abstract
This project develops a system for detecting plagiarism in sets of student assignments written in PHP language. Plagiarism is viewed as a form of code obfuscation where students deliberately perform semantics preserving transformations of an original working version to pass it off as their own. In order to detect such obfuscations we develop a tool in which we attempt to find transformations that have been applied, using several techniques and algorithms. The main goal is to provide fast and quality tool for plagiarism recognition that can be used at academic enviroment.

Keywords: Plagiarism — PHP — Plagiarism detection in PHP code — Code obfuscation — Haelstead metrics — Levenshtein algorithm — Document fingerpring — Winnowing algorithm — Tokenization — Abstract syntax tree

Supplementary Material: Plagiarism Recognizer on GitHub

*xkrpecqt@gmail.com, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Plagiarism is a serious problem not only in educational and scientific institutions and as such it has very long history. International standard ČSN ISO 5127-2003 describes it as ”action or practice of taking someone else’s work, idea, etc. and passing it off as one’s own; literary theft.” In this paper we only focus on plagiarism in coding which is not entirely a new phenomenon. The issue has been discussed and studied previously by researches to identify the severity of the problem and factors which contribute to the act of plagiarism. In programming assignments, plagiarism does not necessarily involve only simple copying of the source but also a code obfuscation. Code obfuscation is the transformation of the source code in such a way that makes it unintelligible to human readers of the code. Obfuscation is a semantics preserving transformation applied to the program source code in the same way as plagiarism. However we must be aware of the following issue. According to Bob Zeidman [1], code transformations can be divided into six categories, but only one of them is considered as plagiarism.

1. **Third-party source code.** It is possible that widely available open source code or libraries are used in both programs. If two different programs use the same third-party code, the programs will be correlated.
2. **Code generation tools.** Automatic code generation tools, such as Eclipse or Netbeans IDE, generate a source code that looks very similar with similar or identical elements.
3. **Common identifier names.** Certain identifier names are commonly taught in schools or commonly used by programmers in certain industries. For example, the identifier *result* is often used to hold the result of an operation. These identifiers will be found in many unrelated programs and will result in these programs being

correlated.

4. **Common algorithms.** In one programming language there may be an easy or well-understood way of writing particular algorithm that most programmers use. As an example we can choose any sorting algorithm. These commonly used algorithms will show up in many different programs, resulting in a high degree of correlation between the programs even though there was no contact between the programmers.
5. **Common author.** It is possible that one author will create two programs that have high correlation simply because that programmer tends to write the code in certain way. Therefore two programs written by the same programmer can be correlated due to style being similar even though functionality of those programs might be completely different.
6. **Copied code.** Code was copied from one program to another, causing programs to be correlated. If all of the previously stated reasons for program correlation has been eliminated, the reason that remains is copying. If the copying was not authorized by the original author, then it comprises the plagiarism.

However detecting plagiarism in educational enviroments brings another issues to the fold. The main problem is that the given student assignments, especially those in beginner's courses are standardized and very strictly specified which may result in similar programs that will be correlated even though students did not work together.

Due to this and other obstacles mentioned before its not possible to simply compare two source codes and if they correlate then accuse authors from plagiarism. That is where most commonly used plagiarism detection tools fail. Most of them are based on standard principle such as remove comments, ignore all blanks and extra lines, except when needed as delimiters and perform a character string comparison between two for all program pairs.

This aproach is not only ineffective, but also very time consuming. Therefore the requirements for our Plagiarism Recognizer (further referred as a PHPR) are a higher. We focus on two specific requirements. First, from complexity point of view, PHPR should recognize plagiarism from other five categories mentioned earlier. To achieve that level of complexity we will use several plagiarsm detection techniques such as Halstead metrics [2], Levenshtein algorithm [3], etc. Second, PHPR should be fast enough to correlate approximately one thousand of student assignments in reasonable time. In order to save time, our tool works in several phases. In the first and second phase assignments are prepared for being correlated. In the third phase the PHPR uses Halstead metrics and Levenshtein algorithm to do shallow analysis of source codes. If any similarity is found, PHPR digs deeper using advanced detection techniques such as document fingerprints [4] or abstract syntax tree comparison [5]. There are many others plagiarism detection techniques such as call graphs and dependency graphs comparison [6] or conceptual similarity [7], that can be implemented in the future as an improvement to the PHPR.

2. Problem analysis

As previously stated, we cannot just simply compare two programs, but we need to expect that plagiarism of program source was achieved either by code obfuscation or by manually applied transformations to a source code from another student.

```
class Restruct {

     private $x = 5;
     private $y = 3;
     private $z = 2;

     // public constructor
     public function Restruct() {
        $this->doThat();
     }

     // function that do this
     public function doThis() {
       echo $this->x . "\n";
     }

     // function that do that
     public function doThat() {
       echo $this->z . "\n";
       $this->doThis();
     }
   }

   $res = new Restruct();
```

Figure 1. A simple class with field and method definitions

These transformations can be both simple such as adding, removing or changing comments, renaming variable names, and complex such as structure changing transformations, dividing program into modules etc. According to Faidhi and Robinson [8] there are six levels of source code transformation to create a plagiarism in order from simple to complex methods.

Level 0 Original program without modifications.
Level 1 Only comments are changed.
Level 2 Identifiers are renamed.
Level 3 Code positions is changed. For instance, field variable declarations are moved from the top of the source file to the bottom.
Level 4 Constants and functions are changed, e.g. in-line procedures.
Level 5 Program loops are changed.
Level 6 Control structures are changed to an equivalent form using different control structure.

```
function doThis($a) {
  echo $a . "\n";
}

function doThat($a, $c) {
  echo $c . "\n";
  doThis($a);
}

$c = 2;
$b = 3;
$a = 5;

doThat();
```

Figure 2. Restructured version of (Figure 1)

This paper is focused on revealing all levels of plagiarism whereby first four levels should be exposed during the shallow analysis of the source codes. Consider an example of simple program restructuring of it's code from object oriented paradigm (Figure 1) to procedural paradigm (Figure 2). On the first sight, both programs look different. Especially if you apply changes on another levels such as the change variable names and comments, but they functionality is exactly the same.

[Code obfuscation] Code obfuscation is the transformation of a source code in such a way that makes it unintelligeble to human readers of the code and reverse engineering tools. Obfuscation is a semantics preserving transformation applied to program source code in the same way as plagiarism. An obfuscator should meet three conditions: functionality must be maintained, resulting code must be efficient and the code must be obfuscated which means that souce code should be changed enough, so reverse engineering is difficult.

Of course making program unreadable is pointless for plagiarism. Therefore, plagiarism using this technique aims to create a semantically identical program with enough implementation differences to confoud plagiarism recognizer tools.

Many obfuscation techniques are commonly used by students as plagiarism techniques. In (Figure 3) we define very simple function. After applying several obfuscation techniques such as adding comments, identifier randomization, modifying variables, we get very different program as shown on (Figure 4).

```
function count($a, $b) {
  return $a + $b;
}
```

Figure 3. Original code before obfuscation

With changes shown in the (Figure 4) standard tool for plagiarism detection using common string comparison method fails for (Figure 4).

3. Static analysis and source code transformations

Static analysis is the analysis of a program code that is performed without actually executing the program. Therefore, during static analysis we perform several operations upon given source code. In first phase we generate JSON object [9] from given assignments. This object is passed to second phase, in which we create all unique pairs of assignments. These pairs are then compared in third and possibly fourth phase as shown in the (Figure 5).

For testing purposes were used some assignments from students of FIT VUT in Brno. Some students wrote two assignments on the same subject, therefore these assignments had much in common. Obviously it is not possible to accuse these authors from self plagiarism, but these assignments are the perfect test data, because we already know that they are similar.

[Tokenization] In the first phase, PHPR is given a set of student assignments and each of them is converted into sequence of tokens. First of all the PHPR removes all whitespaces from source code. Then the basic statistics for each file are created. When tokenization process is completed, the sequences of tokens are split into blocks for the following comparison.

The resulting blocks are then saved into a file as a JSON object that will be used in the next phases. Saving these objects into a file might seem as a waste of time when the other phases follow right after the tokenization, but it is an effective solution in a larger scale. If we want to run the PHPR again and compare old and

```
function fdhf1gr5g4($__dsa45, $cda54fa) {
  return $__dsa45/2 + $cda54fa*1 + $__dsa45/2;
}
```

Figure 4. Obfuscated code

new student assignments, we have the tokenizations process finished at least for old assignments.

In the second phase we generate CSV file containing all unique pairs of student assignments. These pairs are then compared in the third phase and possibly in fourth phase.

[Halstead metrics] is one of the oldest used method in software plagiarism [2]. Halstead defined some basic software science parameters that have been used as a basis for metrics used in plagiarism detection systems. The basic software parameters according to Halstead are: number of unique operands, number of unique operators, number of total occurrences of operators and number of total occurrences of operands.

Halstead uses this to express effort or complexity of program using several equations. Obviously, two programs having the same results in these equations may not be a plagiarism, but they should be inspected further. Therefore choosing correct methods and the correct number of methods is important in obtaining good results.

[Levenshtein distance] is used to compare the similarity of two given strings. This algorithm, also known as string edit distance, takes two strings and gives a value of the distance between them by assigning costs to the action needed to transform one of the strings into another string.

For example the edit distance between the two strings *table* and *bibles* is three because the minimum number of edit operations needed to change word *table* into *bibles* is three: substitute t for b and a for i and insert s at the end. As JSON object created in the first phase contains the subsequences of tokens, we can compare the subsequences using Levenshtein distance.

Both Halstead metrics and Levensthein algorithm is evaluated in the third phase using weighted arithmetic mean. If any similarity between projects is found then these assignments will be investigated in more details during the fourth phase and may be considered as a plagiarism.

[Document fingerprint] is one of the most popular systems used by academic institutions for plagiarism detection. This technique identifies some unique features in a document in order to give it a unique fingerprint. In theory, every different document should have a different fingerprint and similar documents should have similar fingerprints.

For detecting partial copies, we use k-grams. K-gram is a contiguous string of length k. A document is divided into k-grams, where k, ($k > 0$) is the parameter chosen by user. Note that there are almost as many k-grams as the charachter length of the document. Every position in the document marks the beginning of a k-gram. Each k-gram is hashed for efficiency, some subset of these hashes is selected to be the document

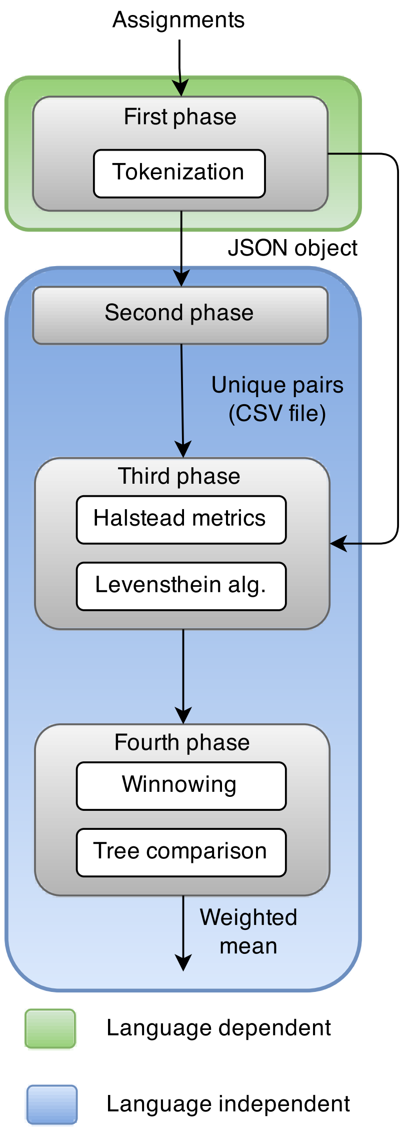

Figure 5. Structure of our PHP Plagiarism Recognizer

fingerprint. A fingerprint also contains positional information, denoting the source code file name and the location within that file that the fingerprint came from. One popular aproach for selecting subset of hashes is to choose all hashes that are *0 mod p*, for some fixed *p*. Where *p* is constant chosen by user. This approach is easy to implement and retains only 1/*p* of all hashes.

As a disadvantage this method gives no guarantee that matches between documents are found because *k*-gram shared between two documents is detected only if its hash *0 mod p*.

Our tool uses winnowing algorithm [4] as one of the best algorithms that implements document fingerprint technique. In this algorithm we define a window of size *w* as *w* consecutive hashes of *k*-grams in a document. By selecting at least one fingerprint from each window algorithm limits the maximum gap between fingerprints. In fact this approach guarantee to detect at least one *k*-gram in any shared substring of length $w + k - 1$.

[Trees] A source code is represented in memory as a parse tree or an abstract syntax tree. Therefore two programs can be compared with tree similarity algorithms providing a more accurate comparison than metric based or string based techniques. For comparison itself, it is probably the best to use Sasha's algorithm [10] for exact and appropriate matching. PHPR goal is to find maximum number of common sub-trees and to calculate a tree edit distance. This technique is very similar to the string edit distance. The tree edit distance between two trees is the cost of the number of operations needed to transform one tree into another.

Unfortunately the standard PHP parser does not create abstract syntax tree from the source code. However, some third-party libraries such as HHVM do, so it is easy to implement the search for plagiarism using tree comparison techniques.

Note that the tree comparison technique and winnowing algorithm are significantly slower than algorithms used in the third phase, so these methods are used only on source codes that might be a plagiarism. This simple selection that allows to omit deeper comparison of assignments with absolutely no similarity is a great way to reduce the computation time of whole PHPR. Results from fourth phase are then evaluated using weighted arithmetic mean.

4. Conclusions

We started out describing some basic facts about plagiarism in academical enviroment and several techniques useful for plagiarism detection. The final version of the PHPR is able to detect plagiarism between the given pairs of PHP programs but results are not totally accurate every time yet. However, it is already useful in practice. Of course the tool only highlights the suspicious assignments and it is user's responsibility to decide whether it is a plagiarism.

As for future improvements, there are still several plagiarism detection techniques that are neither implemented nor mentioned in this paper. In addition the pair generating technique in second phase and comparison techniques implemented in third and fourth phase are general enough to work even for another languages if the intermediate JSON file is delivered.

Acknowledgements

I would like to thank my supervisor Ing. Zbyněk Křivka, Ph.D. for his help and advices.

References

[1] B. Zeidman. What, exactly, is software plagiarism? Intelectual Property Today [blogpost], 2007. On-line at http://www.iptoday.com/pdf/2007/2/Zeidman-Feb2007.pdf.

[2] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science, 1977. ISBN 0444002057.

[3] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8), 707–710, 1966.

[4] D. S. Wilkerson S. Schleimer and A. Aiken. Winnowing: local algorithms for document fingerprinting. Proceedings of the 2003 ACM SIGMOD international conference on Management of Data, 76-85, 2003.

[5] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002. ISBN 3540435506.

[6] J. Krinke. Identifying similar code with program dependence graphs. Proc. Eigth Working Conference on Reverse Engineering, 301-309, 2001.

[7] G. Mishne and M. de Rijke. Source code retrieval using conceptual similarity. Proc. 2004 Conf. Computer Assisted Information Retrieval, 539-554, 2004.

[8] J. A. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computers and Education, 11(1), 11-19, 1987.

[9] T. Bray. The javascript object notation (json) data interchange format. RFC 7159. On-line at http://tools.ietf.org/html/rfc7159.

[10] K. Zhang, D. Shasha, and J. T. L. Wang. Approximate tree matching in the presence of variable length don't cares. Journal of Algorithms, 16, 33-66, 1993.