Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 47–62

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Finding Similarities in Source Code Through Factorization

Michel Chilowicz[1] Étienne Duris[1] Gilles Roussel[1]

Université Paris-Est,
Laboratoire d'Informatique de l'Institut Gaspard-Monge, UMR CNRS 8049,
5 Bd Descartes, 77454 Marne-la-Vallée Cedex 2, France

Abstract

The high availability of a huge number of documents on the Web makes plagiarism very attractive and easy. This plagiarism concerns any kind of document, natural language texts as well as more structured information such as programs. In order to cope with this problem, many tools and algorithms have been proposed to find similarities. In this paper we present a new algorithm designed to detect similarities in source codes. Contrary to existing methods, this algorithm relies on the notion of function and focuses on obfuscation with inlining and outlining of functions. This method is also efficient against insertions, deletions and permutations of instruction blocks. It is based on code factorization and uses adapted pattern matching algorithms and structures such as suffix arrays.

Keywords: program transformation, factorization, inlining, outlining, similarity metrics, software plagiarism

1 Introduction

Current similarity detection techniques on source codes are mainly inspired from computer genomic algorithms [21,23,11] in which factorization and development (outlining and inlining) have not been widely explored. In this paper, we present a new similarity detection technique based on the factorization of source code that permits finding similarities and quantify them at a *function*-level thanks to a synthetized call-graph of the programs.

[1] Email: firstname.lastname@univ-paris-est.fr

doi:10.1016/j.entcs.2009.09.040

This technique consists in splitting each original function into sub-functions, sharing as much sub-functions as possible between original functions. The whole source code is then represented by a call graph: the vertices are the functions and the edges are the function calls. Thus, similar functions should derive in an identical, or comparable, set of sub-functions, i.e., leaves of the call graph. Depending on the measurement used to evaluate (sub-)function similarities, this technique may not be sensible to insertion, deletion, inversion, inlining or outlining of source code which are the most common methods of obfuscation.

The rest of this paper is organized as follows. Section 2 gives an overview of the whole detection process. The main steps of the factorization algorithm and call graph analysis are detailed in section 3 and 4. Related work and benchmarks are discussed in sections 5 and 6. The paper then presents a conclusion and introduces some future work.

2 Overview

Our work is directed towards finding similarities in source code in procedural languages at the level of function. Our aim is to experiment a similarity detection technique based on function transformations. Indeed, to be insensitive to outlining, inlining and source block displacements, we manipulate abstractions of the programs that are call graphs. Initially, the call graph is the one of the original program. Next, by using several algorithms, the call graph is enriched with new synthetic sub-functions that represent the granularity of similarities.

2.1 From source code to token sequences and call graph

From the original source code of a program, a lexical analysis provides us with a sequence of tokens. In this sequence, the concrete lexical tokens are represented by abstract parameterized tokens. Some tokens, such as function parameters or variable declarations are discarded. It is also possible to use normalization techniques, based on syntactic or semantic properties, to filter and enhance the result. For instance, the sequence we process is independent of the identifiers of variables and functions in order to defeat the simplest obfuscations[2]. Since the tokens could be scattered and merged into synthetic sub-functions by the factorization process, each token is associated with its position (project, file, line number of the token) in the source code in order

[2] This first step is also performed by other similarity detection tools [18,12] with abstract tokens.

to backtrace information. A segment tree [8] is used to store this information, and is updated to take into account synthetic sub-functions.

After the normalization and the tokenization step, each function of the source code is represented as a sequence of two kinds of parameterized abstract tokens: a *primitive* token is a reserved keyword of the language, a constant or a variable identifiers and a *call* token, noted $\langle f \rangle$, models a call[3] to a function f. We respectively note Σ^p and Σ^c the sets of primitive and call tokens. For instance, figure 2 gives the token sequence and alphabets obtained from the C function f_2 of the figure 1.

Each original function of the source code being represented as a sequence of tokens, the next step of our algorithm is then to identify common factors in these functions and to *factorize* them in some new *outlined* synthetic functions. However, several factorizations are possible and they potentially require heavy computations. To cope with these problems, we use classical pattern matching data-structures, such as suffix arrays, and some new dedicated structures, such as the Parent Interval Table, allowing our algorithms to be efficient.

2.2 The factorization process

To be able to efficiently identify common factors (identical substrings of tokens) in token sequences of functions, we use the suffix array structure [17]. To minimize the number of synthetic outlined functions, we first seek if a (part of a) small function could be found in a larger one. We note $F^0 = \{f_1, \dots, f_n\}$ the set of all original functions, sorted by increasing length. Thus, for each function $f_k \in F^0$ we scan the functions $F^0_{<k} = \{f_1, \dots, f_{k-1}\}$ (functions whose length are smaller than the length of f_k) to find pairs of factors (u_1, u_2) with $u_1 \in \text{fact}\,(F^0_{<k})$ (u_1 is a factor of a function $f_j \in F^0_{<k}$, i.e., $j < k$) and $u_2 \in \text{fact}\,(\{f_k\})$ such as $u_1 = u_2$. Thereafter, the occurrence u_1 is outlined into a new synthetic function $g = u_1$ whereas the occurrence u_2 from f_k is replaced by the call token $\langle g \rangle$. We note that the function f_j of u_1 is unchanged at this step, to allow larger factors to be identified in a further f_l for $l > k$. The next iteration of the algorithm will allow the entire sequence of g to be identified as common with the factor u_1 of f_j, and the replacement of u_1 by $\langle g \rangle$ will occur at this time. Anyway, if the factor u_1 covers an entire function f_j, there is no need to create a new function g. These operations of search and replacement of factors by call tokens are undertaken for all of the functions from F^0. Thus, for the next iteration, a new set of functions $F^1 \supseteq F^0$ can be deduced from F^0 adding the new outlined functions and ordering the resulting set by increasing length.

[3] We note that function calls may be statically ambiguous in some languages like Java.

```
 1 int min_index(int t[],int start) {                      f3
 2   int i_min, j;
 3   i_min=start; j=start+1;                      Φ1
 4   while (j<SIZE) {
 5     if (t[j]<t[i_min]) i_min=j;
 6     j++;
 7   }
 8   return i_min;
 9 }

10 void exchange(int t[],int a,int b) {                    f1
11   int tmp;
12   tmp=t[a]; t[a]=t[b]; t[a]=tmp;                       Φ2
14 }

14 void subsort(int t[],int i) {                           f2
15   int i_min;
16   if (i>=SIZE) return;
17   i_min=min_index(t,i);
18   exchange(t,i,i_min);
19   subsort(t,i+1);
20 }

21 void inlined_subsort(int t[],int i) {
22   int i_min, j, tmp;                                    f4
23   if (i>=SIZE) return;
24   i_min=i; j=i+1;                              Φ1
25   while (j<SIZE) {
26     if (t[j]<t[i_min]) i_min=j;
27     j++;
28   }
29   tmp=t[i]; t[i]=t[i_min]; t[i_min]=tmp;                    Φ2
30   inlined_subsort(t,i+1);
31 }
```

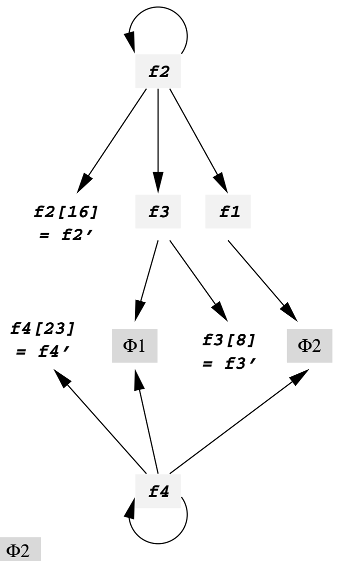

Figure 1. Source code example and call graph deduced after two iterations

Several iterations of this algorithm are needed to achieve better precision for the call graph, in particular to detect shorter redundancies. Nevertheless, at the end of each iteration, we are able to evaluate a score of similarity between two functions. This score is based on the number and the weight of common leaves of the call-graph that are attainable from each of these functions. To ease the evaluation, only two kinds of functions are considered according to their content: either they only contain primitive tokens or only call tokens. A mixed function is transformed into a function containing only call tokens via an outlining of all subsequences of primitive tokens.

2.3 Example

For a better understanding, we introduce a small example explicited on figure 1: a bubble-sort in C where the recursive function `subsort` (f_2) calls the functions `exchange` (f_1) and `min_index` (f_3). The function `inlined_subsort` (f_4) could be considered as an obfuscated version of f_2 after the inlining of f_3 and f_1. Actually, from such a source code, our algorithm first produces a sequence of abstract tokens. For example the dictionaries Σ^p, Σ^c and the raw

Σ^p	Σ^c	Raw sequence of abstract tokens from f_2
`IF, LPAR, RPAR, GE`	`<min_index>`	`IF LPAR identifier GE identifier RPAR RETURN SEMI`
`identifier, ASSIGN`	`<exchange>`	`identifier ASSIGN <min_index> SEMI`
`RETURN`	`<subsort>`	`<exchange> SEMI`
`SEMI`		`<subsort> SEMI`

Figure 2. Tokenization of function f_2 (`subsort`)

sequence of abstract tokens extracted from f_2 are explicited on figure 2. Note that variable declaration tokens have been omitted.

The first iteration of the algorithm searches matches in $F^0 = \{f_1, \ldots, f_4\}$ (we chose function indices with respect to increasing weight of functions, i.e. number of tokens in our example). During the examination of f_4 two matches in f_3 (new function Φ_1) and in f_1 (new function Φ_2) are found. The functions Φ_1 and Φ_2 are outlined and replaced by their call tokens in f_4. Depending on the chosen weight-threshold for match reporting, another function Φ_3 corresponding to the lines 16 and 23 could be outlined. Here we consider instead that these lines yield two distinct primitive-token functions (leaves) $f'_2 = f_2[16]$ and $f'_4 = f_4[23]$. In order to ensure that f_3 only contains call tokens, a new function f'_3 is also outlined, corresponding to the line 8 of f_3: thus, f_3 only contains two call tokens to Φ_1 and f'_3. Finally, at the end of the first iteration, the set of functions ordered by increasing weight is $F^1 = \{f'_3, f'_2, f'_4, \Phi_2, f_1, f_4, f_2, \Phi_1, f_3\}$.

During the second iteration, Φ_2 is found and replaced by a call token while examining f_1 and Φ_1 is found in f_3. We then obtain the call graph explicited on figure 1 at the end of the second iteration.

This graph allows us to define three kinds of metrics, or scores, based on the number and the weight of leaves attainable from each function. We define more precisely in section 4 the score of inclusion (sc_{incl}) of a function into the other, the score of coverage (sc_{cover}) of a function by another one and finally the score of similarity (sc_{simil}) between two functions. For example, for functions f_2 and f_4, we obtain following values: $sc_{incl}(f_2, f_4) \approx 0.88 > sc_{cover}(f_2, f_4) \approx 0.85 > sc_{simil}(f_2, f_4) \approx 0.76$.

3 Factorization algorithm

In this section, we focus on the algorithms and data-structures involved in the factorization of a given function f_i using most-weighted non-overlapping factors of $F_{<i}$. Realistic examples generate long sequences with a lot of tokens. For concision and in order to easily exhibit interesting cases, we will

rank	(f,p)	suffix (not stored)
1	$(f_1,2)$	`a`
6	$(f_2,0)$	`a a b a`
7	$(f_5,2)$	`a a b b a b a b a`
8	$(f_1,0)$	`a b a`
13	$(f_5,0)$	`a b a a b b a b a b a`
14	$(f_4,1)$	`a b a b a`
16	$(f_5,3)$	`a b b a b a b a`
17	$(f_1,1)$	`b a`
22	$(f_5,1)$	`b a a b b a b a b a`
23	$(f_3,1)$	`b a b a`
26	$(f_4,0)$	`b a b a b a`
28	$(f_3,0)$	`b b a b a`
29	$(f_5,4)$	`b b a b a b a`

Figure 3. Suffix array for the example

use a simplistic [4] but illustrative running example with the alphabet $\{a, b\}$ for primitive tokens and with five initial functions of F^0:

$$f_1 = aba \quad f_2 = aaba \quad f_3 = bbaba \quad f_4 = bababa \quad f_5 = abaabbababa$$

We will show how f_5 could be factorized as $aba \cdot ab \cdot bababa$, and then be considered as constituted by sub-functions $f_1\,[0..2]$, $f_2\,[1..2]$ and $f_4\,[0..5]$.

3.1 Working with suffix array

First of all, we have to identify common factors (sub-sequences of tokens) in functions. Indeed, while inspecting the sequence of tokens of f_5, we want to know which parts of others functions are candidates for a match. To deal with this problem, our algorithms rely on a suffix array [17], i.e., an indexed table containing all suffixes of the functions, sorted according to a given total order on the tokens, e.g. lexicographical. Various suffix array construction algorithms may be used such as the direct linear model of Kärkkäinen and Sanders [13]. The main advantage of this structure is its small size since for a string of length n only $\log_2 n$ bits are needed for each suffix whereas the better known construction of suffix tree requires at least 10 bytes by indexed token [16].

Figure 3 gives a representation of the suffix array for our running example, where each suffix appears only once: when the same suffix appears in several functions, we only show in this figure the smallest function index and position, and increment the rank in the array for each occurrence of this suffix. For the sake of clarity, it also shows the corresponding sequences, even if they are actually not stored: a suffix starting at position p of function f_i is simply designated by the pair of integers (i, p). We also need the reverse suffix array,

[4] The example presented in section 2 is more detailed on the following web page: http://igm.univ-mlv.fr/~chilowi/research/finding_similarities/

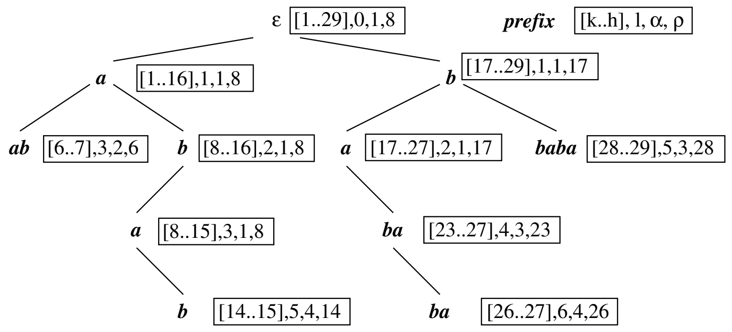

Figure 4. Tree structure represented by the PIT

not represented here: given function index i and a position p defining a suffix of this function, it provides its rank in the suffix array.

3.2 Searching Factors through Parent Interval Table

The key idea of the suffix array structure is that suffixes sharing the same prefix are stored at consecutive ranks. Thus, the notion of rank *interval* makes sense: for instance, all factors starting with aba could be found in the interval [8..15]. Nevertheless, intuitively, we are looking in a "long" function for factors of "shorter" functions. Thus, if we search factors of the complete function $f_5 = abaabbababa$ in the suffix array, we find the interval (singleton) [13..13] corresponding to the function f_5 itself. This is not a suffix of a function of smaller index, and we have to successively consider all shorter prefixes of this sequence until finding a rank interval concerning a function of smaller index.

To efficiently search for a maximal factor, we introduce a new data structure: the Parent Interval Table (PIT). It is deduced from the complete interval tree of the suffix array, i.e., a suffix tree without its leaves. To each suffix s, the PIT associates a set of data $([k..h], l, \alpha, \rho)$. In the suffix array, k and h ($k \neq h$) are the ranks of the first and the last suffixes that share the longest common prefix (lcp) with s; l is the length of this lcp of the interval $[k..h]$. Finally, α is the minimal index of the functions associated to its interval, and ρ is the rank of the longest suffix extracted from the function of index α in the interval. Figure 4 shows the structure represented by the PIT for our example.

To find the maximal-lcp interval in the suffix array, different strategies may be used. The simplest is to build the complete interval tree of the suffix array, saving the parent pointers between intervals. Another way is to save only the *needed* intervals. An interval is needed if and only if its α-value is strictly smaller than the α-value of one of its children. We use a dedicated algorithm (not detailed here) to compute the PIT and the needed intervals on the branches thanks to the LCP tables, already computed in linear time [22].

This algorithm uses a simple sequential reading of the LCP table with a stack simulating the current branch of the interval tree. It runs in linear time, with a memory complexity in the height of the interval tree (linear in the worst case).

Our algorithm identifies all the maximal factors of a function f_i that match with parts of functions of $F_{<i}$. Thus, for each position p in f_i, it looks for a factor of a function in $F_{<i}$ sharing the longest prefix with $f_i\,[p..]$. To find it, the information $([k..h], l, \alpha, \rho)$ is retrieved from the PIT for the suffix $f_i\,[p..]$:

- If $\alpha < i$, then ρ is the rank in the suffix table of a function (f_α, q) such as $f_\alpha\,[q..q+l-1]$ is a maximal factor of $f_i\,[p..]$. For instance, for $f_5[3..]$, (*abbababa*), the PIT yields the information $([8..16], 2, 1, 8)$: since $\alpha = 1 < 5$ and $\rho = 8$ is the rank of $(f_1, 0)$ we found $f_1\,[0..0+2-1]$ (*ab*) as a maximal factor of $f_5[3..]$.
- If $\alpha \geq i$, and since we are only looking for factors in $F_{<i}$, we must examine the parent interval of smaller *lcp*, i.e., a parent node in the PIT, until finding an interval whose minimal function index is smaller than i. For instance, looking for maximal factor of $f_4[0..]$ (*bababa*) leads to $([26..27], 6, 4, 26)$: since $\alpha = 4 \geq 4$, we have to consider the interval of the parent node, $([23..27], 4, 3, 23)$. Now, $\alpha = 3 < 4$ and $\rho = 23$ being the rank of $(f_3, 1)$, we found $f_3\,[1..1+4-1]$ (*baba*) as a maximal factor of $f_4[0..]$.

3.3 Eliminating overlapping factors

Once a set of maximal factors is associated with f_i, one for each position, overlapping between them is expected. For the example of f_5, these factors appear in the three left-most column of the table of figure 5. Furthermore, several combinations of (parts of) these factors are allowed to cover the function f_i. For instance, f_5 could be covered by *ab.aa.bbaba.ba* or by *aba.ab.bababa*.

To retain a factorization of f_i using most-weighted [5] non-overlapping factors, all the matched factors are placed in a priority queue, and the most-weighted ones will be successively selected, as indicated by the numbered arrows in the figure 5. For each selected factor, we must remove all intersecting parts from all other candidate factors before selecting the next factor.

To achieve this, a segment tree [8] is created and fed with the intervals

[5] The weight function is defined on Σ^*. In a first approach we define the weight function $w(u \in \Sigma^*) = w(u_0) + w(u_1) + \cdots + w(u_{|u|-1})$ and a constant weight vector associating to each primitive token a constant value. To infer the weights of the call tokens, we establish for each function f the set of reachable leaf functions $lf(f)$ according to the call graph of the previous iteration (see 4.1) and compute the sum of the weights of these reachable leaf functions. Thus the weight of a call token reflect the amount of coverage of code of its underlying called function.

suffixes of f_5	factors in $F_{<5}$	abaabbababa	abaabbababa	abaabbababa	factorization
$f_5\,[0..]$	$f_1\,[0..2]$	aba	**aba** $\leftarrow 2$	**aba**	**aba**
$f_5\,[1..]$	$f_1\,[1..2]$	ba	ba	ba	ba
$f_5\,[2..]$	$f_2\,[0..2]$	aab	aab	a**ab** $\leftarrow 3$	a**ab**
$f_5\,[3..]$	$f_1\,[0..1]$	ab	ab	ab	ab
$f_5\,[4..]$	$f_3\,[0..4]$	bbaba	bbaba	bbaba	bbaba
$f_5\,[5..]$	$f_4\,[0..5]$	**bababa** $\leftarrow 1$	**bababa**	**bababa**	**bababa**
$f_5\,[6..]$	$f_4\,[1..5]$	ababa	ababa	ababa	ababa
$f_5\,[7..]$	$f_3\,[1..4]$	baba	baba	baba	baba
$f_5\,[8..]$	$f_1\,[0..2]$	aba	aba	aba	aba
$f_5\,[9..]$	$f_1\,[1..2]$	ba	ba	ba	ba
$f_5\,[10..]$	$f_1\,[2..2]$	a	a	a	a

Figure 5. Selection of non-overlapping matches on f_5

of positions of the factors found on f_i. A segment tree is an adaptation of a balanced binary tree: each node is a interval whose key is the low bound. This structure can be used to find k intersecting factors in $O\left(\max(k, \log|f_i|)\right)$. For each iteration we extract the most-weighted factor u from the priority queue and search all the intersecting factors $V = \left\{v_1, \ldots, v_{|V|}\right\}$ in the segment tree. Since we use an homogeneous weight function[6], u is not a strict factor of any factors of V ($\forall v \in V, w(u) \geq w(v)$). For each factor $v_k \in V$, we remove it from the priority queue and from the segment tree. If it is not a factor of u but shares an intersecting part on the left or on the right, we remove the intersecting part of v_k with u to obtain v'_k that is added in the segment tree.

Since there is at most one match for each position in f_i, the most-weighted matches must be selected between at most $|f_i|$ matches. For each selected match there is at most $|f_i| - 1$ intersecting factors, each of them needed a time in $O\left(\log|f_i|\right)$ to update the segment tree and the queue. So the global time complexity is in $O\left(|f_i|^2 \log|f_i|\right)$ ($O\left(|f_i| \log|f_i|\right)$ if we use the length of the factor as a weight function).

4 Call graph analysis

The process of factorization we apply is based on the algorithm previously described, which is iterated several times. Nevertheless, each iteration provides us with a call graph whose nodes are functions, with leaves being primitive functions (composed of only primitive tokens). This graph could be used to define and measure some metrics, or scores, relative to similarity between two functions. We will first define abstract metrics qualifying similarity between

[6] A weight function w is homogeneous iff $\forall(x, y) \in \Sigma^2$ such as $x \in \text{fact}\,(y)$, $w(x) \leq w(y)$.

two projects, and then present how we value them in the concrete context of our process. Next, we will see that these metrics could be used to enhance the factorization algorithm at each iteration.

4.1 Quantifying plagiarism

To measure the degree of similarity on pairs of projects (a project being one or a set of functions), we define an abstract *information* metrics $I(p_i)$ representing the amount of information contained in a project p_i. The associated metrics $I(p_i \cap p_j)$ and $I(p_i \cup p_j)$ quantify respectively the amount of common information shared between two projects p_i and p_j and the total amount of information contained in the two projects p_i and p_j. We then define the three following metrics over projects:

- $sc_{incl}(p_i, p_j) = \frac{I(p_i \cap p_j)}{\min(I(p_i), I(p_j))}$ is the *inclusion* metrics and states the degree of inclusion of one project into another. In fact $sc_{incl}(p_i, p_j) = 1$ means that the smaller project (in term of amount of information) is included into the greater.
- $sc_{cover}(p_i, p_j) = \frac{I(p_i \cap p_j)}{\max(I(p_i), I(p_j))}$ is the *coverage* metrics and quantifies the degree of coverage of the greater project by the smaller project.
- $sc_{simil}(p_i, p_j) = \frac{I(p_i \cap p_j)}{I(p_i \cup p_j)}$ is the *similarity* metrics and measures the degree of similarity between the two projects p_i and p_j. We note that $sc_{simil}(p_i, p_j) = 1$ iff the projects p_i and p_j are identical according to the information metrics I.

Since $I(p_i \cup p_j) \geq \max(I(p_i), I(p_j)) \geq \min(I(p_i), I(p_j)) \geq I(p_i \cap p_j)$ the following inequality is verified: $sc_{incl}(p_i, p_j) \geq sc_{cover}(p_i, p_j) \geq sc_{simil}(p_i, p_j)$.

In the particular case of our algorithm, we evaluate concrete metrics based on the previous definitions. Considering the graph of call relations deduced from the last iteration of the factorization algorithm, the set $lf(f)$ of leaf functions (functions of primitive tokens) reached by each internal function f (not leaf) is computed by transitive closure. Then, for each pair of internal functions $(f1, f2)$ the set $lf(f1) \cap lf(f2)$ is computed. Based on the weight function w on leaf functions ($w : f \longrightarrow |f|$ may be used), we define the function W that associates to a set of leaf functions the sum of their weights. These data are used to introduce a concrete definition of the amount of information I contained in projects or shared between functions:

$$I : \begin{cases} I(f) = W(lf(f)) \\ I(f1 \cap f2) = W(lf(f1) \cap lf(f2)) \\ I(f1 \cup f2) = W(lf(f1) \cup lf(f2)) \end{cases}$$

Then we deduce a valuation for the metrics sc_{incl}, sc_{cover} and sc_{simil}:

$$\begin{bmatrix} sc_{incl} \\ sc_{cover} \\ sc_{simil} \end{bmatrix} : (f1, f2) \longrightarrow W\left(lf(f1) \cap lf(f2)\right) \cdot \begin{bmatrix} 1/\min\left[W\left(lf(f1)\right), W\left(lf(f2)\right)\right] \\ 1/\max\left[W\left(lf(f1)\right), W\left(lf(f2)\right)\right] \\ 1/W\left(lf(f1) \cup lf(f2)\right) \end{bmatrix}$$

These metrics have different purposes. sc_{incl} is useful to quantify plagiarism: if $sc_{incl}(f1, f2) = 1$, then one of the internal functions shares all its reachable leaf functions with the other like in case of small functions copied from larger ones or the contrary. For instance, this score detects a plagiarism between the functions f_4 and Φ_1 of figure 1. sc_{cover} quantifies the amount of coverage of the leaf functions of the heaviest function (in terms of the sum of the weights of the reachable leafs) by those of the lightest. Finally $sc_{simil}(f1, f2)$ measures the amount of similarity between $f1$ and $f2$: $sc_{simil}(f1, f2) = 1$ iff $lf(f1) = lf(f2)$.

Since reachable functions are represented by sets, without order consideration in the defined family of metrics, two algorithmically distinct functions may be considered similar. This method may report false-positive matches, in particular if the length of leaf functions is not lower-bounded[7].

For a graph of n functions, computing the weight vector and the similarity matrix requires the knowledge of the reachable leafs of each function (determined in $O(n^2)$ through a graph-walk) and the computation of the intersection of the reachable leaves of each pair of function (in $O(n)$). Finally, the computation of a similarity matrix between all functions takes a temporal complexity in the worst case of $O(n^3)$.

4.2 Clustering of similar functions

After each iteration, we introduce a clusterization process on the internal functions to obtain equivalent classes of functions. Each call token is replaced by a member of its equivalence class. This induces a simplification of the call graph with the removal of other functions of the class and some unshared leaf functions. In the example of figure 1, f_2 and f_4 may be clustered into an equivalent class with the removal of $\{f_2', f_3'\}$ or $\{f_4'\}$ depending on the removed function (f_2 or f_4). For the next iteration a new total order on Σ^c is deduced. Here a call token to f_2 or f_4 will be considered equivalent. For instance, if two sequences $\langle f_2 \rangle \langle f_2 \rangle \langle f_4 \rangle$ and $\langle f_4 \rangle \langle f_4 \rangle \langle f_2 \rangle$ are encountered, they cannot be reported as a match. But if f_2 and f_4 are sufficiently similar to be clustered, they will be renamed and the two sequences will exactly match at the next iteration. Clustering introduces false-positives but increases the recall.

[7] If the leaf functions are reduced to primitive instructions all functions may be similar.

We also consider a clusterization process on the leaf functions in order to treat as similar near sequences of primitive tokens to cope with local additions, deletions or substitutions of tokens.

5 Related work

Many research work has already focused on the detection of similarities. Among them, some look for similarities in free texts [7], in this section, however, we will focus on research work that search similarities in source code. These work may be divided into two main groups depending on their aim.

The first group of work focuses on software engineering. It attempts to find exact matching in order to factorize redundant cut-and-paste or to follow the evolutions between different versions of one project. Usually they do not address the problem of obfuscation. In this context exact methods of pattern matching [1,9,3] and complex algorithms on the abstract syntax trees [2] or the dependency graphs [10,15] may be applied. Several tools such as CCFinder [4] or CloneDR [6] are used to implement these approaches.

The second group of work focuses on plagiarism detection. Its objectives are to detect similarities in the potential presence of intentional obfuscation. The oldest of these works [20,25] that use different program metrics to compare whole programs are usually defeated by simple transformations. Consequently, techniques have been used to get around this problem. Several of these techniques [21,23] are based on Karp-Rabin fingerprints of token n-grams [14]. Two popular applications accessible on the Web are used to implement these approaches. Moss [18] selects and stores fingerprints in a database through the Winnowing process [23] in order to compare it to a large number of programs. JPlag [12] extends matching code zones for identical fingerprints with the Greedy String Tiling algorithm [26]. SourceSecure [19] considers alignment of semantic fingerprints of salient elements in code generated by grammar-actions rules. Another approach [5] implemented by the web-service SID [24] is based on the LZ compression technique [27] to detect redundant code. As far as we know, the methods based on abstract syntax trees have not been used to address the detection of plagiarism.

Aside from the differences in the algorithms, unlike other tools that consider complete source code files for comparison, our method is based on a directed graph representing the program where similar parts of the code are merged into a single node.

6 Tests

We tested our similarity detection method on a set of 36 student projects (about 600 lines of ANSI-C code for each project) by comparing it against Moss [18] and JPlag [12]. Like Moss and JPlag we work on the raw token stream extracted from the bodies of functions without any preliminary normalization step by syntactic or semantic interpretation.

Among the student projects, we selected a random project that is humanly obfuscated using the following methods: no change (1), identifier substitutions (2), changes of positions of functions in the source code files (3), regular insertions and deletions of useless instructions (4), one level of inlining of functions (5), outlining of parts of functions (6), insertions of identical large blocks of code (7). We also added a non-plagiarized project (8) according to a human review.

We studied top inclusion scores from the generated similarity matrices via Moss, JPlag and our method after 5 iterations using the plagiarism metrics sc_{incl}. Ideally the inclusion (or plagiarism) score between an original project and an obfuscated project based on the totality of code of it should be 1. The plagiarism scores for the different kinds of obfuscation can be found in figure 6 and the execution times (on a Pentium 4 3 Ghz CPU with 1 Go RAM using the Sun JRE 1.5) for our method in figure 7; since Moss and JPlag are server-side executed their running time can not be compared. We note that the parameters used (length of the fingerprinted n-grams and window size) by the Moss online-service are not known ; concerning JPlag the token length threshold was set to 10. For our factorization method we used a token length threshold of 10 to report a match and ignored leafs whose length is smaller than 10 for the computation of leaf sets.

We observed that almost all the studied student projects contained shared chunks of code potentially raising false-positive reporting problems. These can be avoided by discarding code shared by more than one fixed number of projects (here 10).

Our method, in spite of being directed towards the detection of inlining and outlining obfuscation, presents lower similarity scores on these kind of obfuscation than JPlag; the plagiarism score computed by JPlag is not docu-

Kind of obfuscation	1	2	3	4	5	6	7	8
Factorization (sc_{incl})	1.0	1.0	1.0	0.72	0.87	0.81	0.84	0.04
Moss	0.94	0.94	0.90	0.25	0.74	0.56	0.73	0.01
JPlag	0.99	0.99	0.97	0.37	0.87	0.86	0.81	0.00

Figure 6. Similarity scores between the original and the obfuscated projects

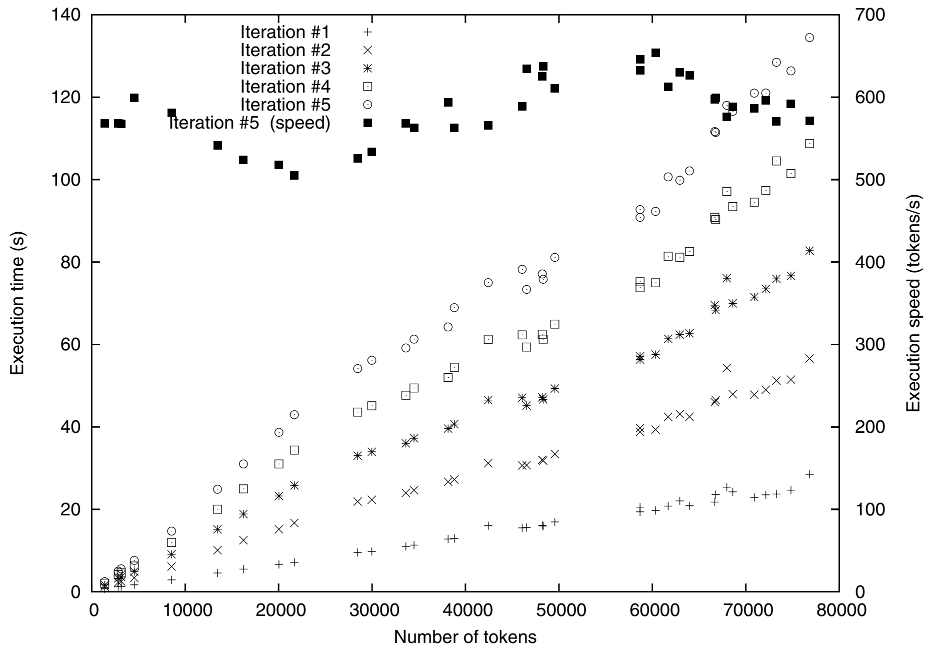

Figure 7. Factorization algorithm running times

mented and we will need more work to explain this difference. However, results are promising for the detection of code obfuscated by insertion and deletion of instructions. The temporal experimental complexity appears linear with the number of initial tokens.

7 Conclusions

Our factorization method for finding similarities in source code shows modest but promising results on preliminary tests. Its main advantage relies on the resistance against obfuscatory methods such as inlining, outlining or shift of blocks of code. We discuss here the limits of our approach and future developments to be considered.

Extreme outlining over the set of functions of projects could result in few short leaf functions (especially when using abstract tokens). For example all of the instructions and assignments could be outlined in leaf functions. Since our method does not consider the order of called functions, the precision is reduced. Some false positives may appear. This is why an upper threshold for the lengths of the leaf functions must be set. Considering new metrics for the comparison of pair of functions that takes into account the order of function calls may be envisaged. Furthermore, some data-flow analysis could be performed on specific parts of the code to reduce false positives.

Currently, our method does not deal with function calls with one or more function calls as argument. This type of situation is not unusual in source code and raises interesting schemes for obfuscation. A preliminary approach could add temporary local variables to unfold the composed calls into intermediate assignments.

Presently, results of outlining processes are viewable as partial outlining graphs where leaf functions are selected according to their attainability from initial functions. Similarity between functions or clusters are used to locate similarities. A more human-friendly tool for the render of results remains to be studied.

References

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*. IEEE CSP, 1995.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*. IEEE-CS, 1998.

[3] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM*, Montreal, October 2002. IEEE-CS.

[4] CCFinder. http://sel.ics.es.osaka-u.ac.jp/cdtools/index.html.en.

[5] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Trans. Information Theory*, jul 2004.

[6] CloneDR. http://semdesigns.com/.

[7] Paul Clough. Old and new challenges in automatic plagiarism detection. National Plagiarism Advisory Service, 2003.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, second edition*. MIT Press and McGraw-Hill, 2001.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, 1999.

[10] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Lang. and Sys.*, 12(1), 1990.

[11] Robert Irving. Plagiarism and collusion detection using the Smith-Waterman algorithm, 2004.

[12] JPlag. https://www.ipd.uni-karlsruhe.de/jplag.

[13] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. ICALP*, volume 2719 of *LNCS*, 2003.

[14] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2), 1987.

[15] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE*, 2001.

[16] S. Kurtz. Reducing the space requirement of suffix trees. *Soft. Practice and Exp.*, 29(13), 1999.

[17] U. Manber and G. Myers. *Suffix arrays: a new method for on-line string searches*. Soc. for Industrial and Applied Math. Philadelphia, PA, USA, 1990.

[18] Moss. http://theory.stanford.edu/~aiken/moss.

[19] Mohamed Amine Ouddan and Hassane Essafi. A multilanguage source code retrieval system using structural-semantic fingerprints. *Int. Journal of Comp. Sys. Sci. and Eng.*, 1(3), 2007.

[20] Alan Parker and James Hamblen. Computer algorithms for plagiarism detection. *IEEE Trans. on Educ.*, 32(2), 1989.

[21] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarism among a set of programs JPlag. *Journal of Univ. Comp. Sci.*, 8(11), November 2002.

[22] Claus Rick. Simple and fast linear space computation of longest common subsequences. *Inf. Process. Lett.*, 75(6), 2000.

[23] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In ACM Press NY, editor, *Proc. Int. Conf. on Management of Data*, 2003.

[24] SID. http://genome.math.uwaterloo.ca/SID/.

[25] Kristina L. Verco and Michael J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In John Rosenberg, editor, *Proc. the First Australian Conf. on Comp. Sci. Educ.*, Sydney, July 3-5 1996. SIGCSE, ACM.

[26] Michael J. Wise. Running karp-rabin matching and greedy string tiling. Technical Report 463, Dep. of Comp. Sci., Sidney Univ., March 1994.

[27] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3), May 1977.