Fast Approximate Matching of Programs for Protecting Libre/Open Source Software by Using Spatial Indexes

Arnoldo José Müller Molina and Takeshi Shinohara
Kyushu Institute of Technology
Department of Artificial Intelligence
820-8502 Fukuoka-ken Iizuka-shi, Kawazu 680-4, Japan
arnoldo@daisy.ai.kyutech.ac.jp, shino@ai.kyutech.ac.jp

Abstract

To encourage open source/libre software development, it is desirable to have tools that can help to identify open source license violations. This paper describes the implementation of a tool that matches open source programs embedded inside pirate programs. The problem of binary program matching can be approximated by analyzing the similarity of program fragments generated from low-level instructions. These fragments are syntax trees that can be compared by using a tree distance function. Tree distance functions are generally very costly. Sequentially calculating the similarities of fragments with them becomes prohibitively expensive. In this paper we experimentally demonstrate how a spatial index can be used to substantially increase matching performance. These techniques allowed us to do exhaustive experiments that confirmed previous results on the subject. The paper also introduces the novel idea of using information retrieval techniques for calculating the similarity of bags of program fragments. It is possible to identify programs even when they are heavily obfuscated with the innovative approach described here.

1 Introduction

Presently, the most popular open source software license is the General Public License (GPL). Individuals are allowed to copy, study, modify and redistribute software with this license. When the binary form of the software is distributed, the corresponding source must be accessible. Otherwise a GPL violation[4] occurs. Other open source licenses have the same restriction.

Our main objective is to create a tool capable of detecting these violations. Specifically, our goal is to receive a *pirate program* (suspected to contain libre software in it) as a query, and return to the user a list of n *candidates* (libre software) that are likely to be embedded in it. Once the candidates have been found, a more detailed and intensive analysis can be conducted. We want to protect as many libre programs as possible at the same time. In other words, we want to build an information retrieval system for programs. As explained in section 6.2, techniques like extracting the "strings" of a program or partitioning the program into blocks ("comparator"), can be easily obfuscated. Our techniques must work even when the binary programs have been obfuscated.

Recent research on approximate program matching [30] showed promising results. The technique works by partitioning a program into *fragments*. Those fragments become a fingerprint of the program. In previous work [30], we incorrectly called these fingerprints "slices". As we shall see in section 2, the term fragment is more appropriate. Fragments are expressions that can be represented by a tree. We use the terms fragment and tree interchangeably. The matching between two programs is achieved by comparing two bags of trees with a tree edit distance function d. The idea is to find pairs of similar trees between the bags. Once these pairs have been found, a score based on the frequency of the matched fragments is calculated. This score indicates the degree of similarity of two programs.

The generally accepted similarity measure for trees is the tree-edit distance[37]. This distance function is very expensive to calculate. Other distance functions [42, 30] are faster, but less precise. Even when these faster functions are used, matching all the trees in a database of programs becomes prohibitively expensive. Filtering is necessary to reduce the amount of distance computations required.

A *spatial database* is a database that can store and query data related to objects in space such as points, lines and polygons. A *spatial index* is used to optimize queries on spatial databases. The *R-tree*[22] is a typical spatial index. *Nearest Neighbor Search* is a numerical optimization problem for finding similar points in multidimensional metric

spaces. A *k-nn* search is a nearest neighbor search that finds the k closest elements to a given object. A *range* r can also be used to get objects that are r distance units away from the query.

The main contribution of this paper is to show how a spatial index improves program matching performance when combined with a tree-edit distance function. A dimension reduction technique called SMAP is used to achieve this. This allowed us to do extensive experiments. Predictions derived from our previous empirical results[30] have been confirmed. The previous ranking technique has been changed to produce better results.

We compare two indexing techniques with different parameters. One of the methods is based on a spatial index (R-tree) and the other on sequential search. We experimentally demonstrate that for small ranges and for small k, a spatial index can deliver the best performance. For bigger ranges, the spatial index method loses its effectiveness against the second method. We also compare two different program ranking techniques and empirically validate their effectiveness.

In this research we have not considered code segment encryption obfuscation techniques. The binary must be unencrypted before we can analyze it.

Section 2 gives background information on how fragments are extracted from a binary program. Section 3 describes the distance function used to match trees. Sections 4 and 5 explain respectively, the techniques used for matching trees and the experimental results obtained. In section 6 we detail some related research. Section 7 summarizes the results of this paper and indicates future directions.

2 Program Fragments

An SSA graph is a program representation. Each node contains sequences of *assignment statements*. The last statement of a node can be a control flow statement (jump, goto). Each variable is assigned exactly once. If a variable has more than one assignment, a new sub-indexed version of the variable is created. Figure 1 shows how the function f may be represented in SSA form. The variables *res* and *count* have a subindex that uniquely identifies each assignment. The ϕ function selects one of its parameters depending on the previously executed basic block.

Initially, a binary program is converted into SSA form[18]. The transformation from a binary program to an SSA representation is outside the scope of this paper. A detailed explanation can be found here[25]. The next step is to take all the assignment expressions within each basic block of the SSA graph and generate fragments from them.

Fragments are generated by taking all the right hand side expressions of all the assignments in the SSA graph. Then, we recursively copy the right hand side of all the references

```
f(int i){
  int res = 1;
  int count = 1;
  while(count <= i){
    res = res × count;
    count++;
  }
  return res;
}
```

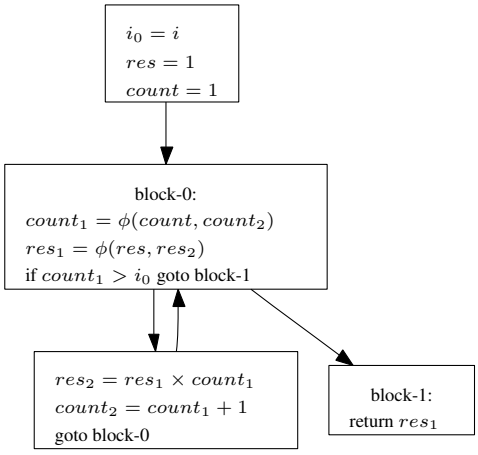

Figure 1. SSA representation and its corresponding source code

to other variables. We expand a variable only once to prevent infinite expansions. Note that a fragment of three nodes corresponds to one machine instruction with two operands. As an example, the fragment for variable $count_1$ in figure 1 is $\phi(1, +(count_1, 1))$. Expressions that cannot be expanded "enough" are discarded because very short expressions are too common to be useful. In our experiments we have observed that trees smaller than five nodes are not useful and affect the results negatively. This is analogous to what information retrieval systems do when they search natural language. In English, very common words like "the" or "an" are ignored. In a similar way, small fragments are too common to be useful.

Our fragment extraction process is described in more detail here [30]. In this paper we incorrectly used the term "slice" instead of fragment. The original definition introduced by Weiser [47] stated that slices are complete programs. It is obvious that fragments do not satisfy this property as they ignore control flow information.

Note that our fragments can be extracted from any representation that can abstract a binary program into a series of assignment expressions. Control flow information is not required. In practice, an SSA graph is convenient because different static analysis can be done before creating the fragments. If an obfuscation prevents our technique

from abstracting the program into a series of assignments, a dynamic fragment extraction approach could be applied.

As a general rule, fragments not modified by some obfuscator, might be modified by another obfuscator. Taking this into account, we can make three observations of semantics preserving transformations. First, we can insert instructions that do not affect the original assignments. New fragments will be created, but the original fragments will remain the same. We also can insert branches that will never be reached. Therefore it is possible to assign *garbage* to any local variable as many times as we want without affecting the semantics of the program. This transformation will create huge ϕ expressions that will never be matched by the techniques presented here as the distance can grow as much as the obfuscator developer wants. The parameters of this ϕ function will have to be modified too, otherwise those fragments will be matched independently. Modifying all the fragments of a method could create huge binary executables.

The second observation is that if portions of the binary code are detected to be unreachable they can be deleted by the obfuscator or compiler. If the obfuscator or compiler uses information contained in the original source code of the program, our static analyzer might not be able to remove those fragments. If the obfuscator/compiler uses only binary-level information then a good static analyzer might be able to delete those fragments too. This attack can only be prevented with a robust static analyzer.

The third observation is that a combination of insertions and deletions of instructions, can replace sequences of instructions. For instance $a \times 2$ could be replaced by $a + a$. Some of those transformations can be normalized automatically. Some can be created by hand using expert knowledge. Others will require more special techniques like the ones described by Walenstein *et al.*[46]. Even if the normalization fails to completely reconstruct the original expression, the tree distance (section 3) might be lower or equal than r, allowing the match. Good term rewriting rules and a good distance function are ways to reduce the damage caused by this attack.

We believe that creating big ϕ functions is the obfuscation which can cause more damage to our technique.

We are currently exploring other fragment extraction strategies that are less vulnerable to this attack.

3 Fragment Similarity

As fragments are expressions, they can be represented by trees. To match fragments, we use a tree distance function. The generally accepted similarity measure for trees is the tree-edit distance introduced by Tai[37]. This problem is known to be NP-hard for unordered trees. The algorithms available for ordered trees are at least $O(n^3)$.

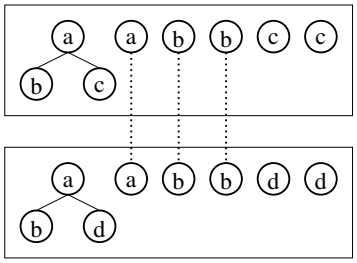

Figure 2. Fragment Similarity

We implemented the algorithm presented by Weimann *et al.* [19]. Even when using Yang's [48] filtering approach the function was not fast enough. Instead, we implemented a distance heuristic that mimics the behavior of the tree-edit distance function. Let E be the set of all fragment expressions. The function tl takes as an input a tree t. It returns a multi-set of all subtrees that compose t including t. Also for each included node, a copy of the element without children is added. We add this to balance the case where the node exists in both trees, but the parameters differ. The structural length $l : E \rightarrow \mathbb{N}$ of a fragment s is $l(s) = |tl(s)|$. Algorithm 1 defines the d function. Because the function tl always returns an even number of elements, the resulting value from d is always in $\mathbb{N}$. The *size* of a fragment s is $\frac{l(s)}{2}$.

An example of tl for trees $a(b, c)$ and $a(b, d)$ can be found in figure 2. Each box represents the sets created by tl for each of the input trees. The arrows show the elements that will belong to the intersection operation internally performed by d. Calculating d for both trees returns:

$d(a(b, c), a(b, d)) = \frac{(6+6)-(2\times 3)}{2} = 3$

Note that a, b, c and d are not variable identifiers. They are low-level machine instruction codes or operands. The algorithm that extracts the fragments from the SSA graph guarantees this. This function has a worst-case complexity greater than $O(n^2)$, but smaller than $O(n^3)$.

Note that d returns 2 for two fragments like "$+(a, b)$" and "$*(a, b)$". They are syntactically close but semantically different. If $r \leq 2$ both fragments will be considered equal by the program ranking techniques described in 4.3.

Algorithm 1 d function

$d : E \times E \rightarrow \mathbb{N}$

$d(e1, e2) = \frac{(l(e1)+l(e2))-(2\times|tl(e1)\cap tl(e2)|)}{2}$

Where $\cap$ is the multi-set intersection operation.

4 Matching Bags of Fragments (Programs)

This section explains the techniques used to match programs. In the following, we describe how spatial indexes

can be used to improve performance. In section 4.2 we describe the sequential search employed. Section 4.3 details two similarity measures for bags of fragments.

4.1 Spatial Indexes

In Euclidean metric spaces, approximate retrieval is efficiently realized by the spatial indexing method R-tree [22]. Trees cannot be directly indexed by a spatial index. Shinohara and Ishizaka[35] introduced a method called SMAP to index objects in any metric space.

To use a spatial index, a tree must be mapped into a tuple. This can be achieved by first selecting i pivots $p_1 \dots p_i$ from the database. The tuple of an object o will have the form $(d(o,p_1), d(o,p_2), \dots, d(o,p_i))$. Of course, other properties like q-grams[32] or node depths [24] can be added to the vector if they are compatible with d. Once the vector is created, the trees can be stored in a spatial indexing method. For this technique to work, it is necessary to configure the R-tree to use the Chebyshev or L_∞ distance. This distance will compute a lower bound of d. If this bound is smaller or equal to r then the tree distance function can be computed. In this way, the number of computations of d is reduced considerably. Current pivot selection strategies help to reduce the amount of computations of d by 95%. See section 5.1.3 for more details on this. SMAP is a relatively new technique, pivot selection strategies are continuously being developed. As these strategies improve better performance may be achieved.

This technique can be used with any distance function that satisfies the triangular inequality. The spatial index we used is a packed R-tree. To our knowledge, this is the first time SMAP is used to match trees.

4.2 Sequential Matching

The sequential matcher takes each of the fragments of the query and matches them against the database. A "tree size" filter was employed. The filter only matches trees from the database whose size is in the range $[min - r, max + r]$, where min is the size of the smallest tree from the query and max is the size of the biggest tree of the query. Additionally, if the sizes of the fragments differ by more than r, we discard the pair as we know that the real distance is greater or equal than r.

It is not scalable to load into memory all the trees from the database at the same time. What we did to overcome this difficulty was to load all the trees from the query in memory and match each object of the database against the query. A priority queue can be used to store partial results. Parsing a tree from its string representation has an important effect in performance. "Tree-caching" is required to minimize this important performance bottleneck. The idea is to keep in memory frequently accessed trees.

This method consumes very little memory and takes advantage of sequential access of files and of memory locality as the entire query can fit in cache (CPU) memory.

4.3 Ranking

In previous work [30] the scoring used for an application A and a query Q was a naive calculation: $\frac{|Q \cap A|}{|A|}$ where two fragments s_1 and s_2 are considered equal if $d(s_1, s_2) \leq r$. We will call this ranking "NR". As section 5.2 details, by using this we were not able to achieve reliable results. We wanted to match programs by taking into account the rareness of their fragments. Also, we wanted to consider the distribution of the fragments (frequency) in the query. Such techniques have been explored for years by information retrieval researchers[8]. We assume that a bag of fragments is analogous to a text document and a word in a document is analogous to a fragment.

Information retrieval systems use techniques like stemming to normalize natural language words to a common base "root". This helps to improve retrieval results. This stemming is analogous to the technique introduced in section 3. What the distance function d allows is to associate a possibly obfuscated fragment to a fragment that is in the database. Fragments stored in the database can be considered analogous to a word's root.

When a database of libre programs is to be constructed, fragments are stored in the spatial index. Additionally, the information of what fragments belong to which documents (programs) is saved into an information retrieval system. Once we receive a query, we use d to find similar fragments in the database within a range r. The next step is to create an artificial document with the fragments obtained in the previous step. This document can be then fed into an information retrieval system as a query and the top n returned documents will be the candidates. When a web search engine allows the user to find pages similar to another page, an analogous procedure is taking place. In this paper, we call the previously described ranking "IR".

Note that information on where the fragments are located (module, function) is never stored. Contrary to what we stated in [30], this is key in protecting our technique from in-lining attacks. An in-lining attack will change the position of the fragments, and their frequency. The first problem is irrelevant for our matching method, and the second one can be compensated by using a robust scoring technique.

For this research, we employed an open source information retrieval software called Lucene[1]. As an efficiency consideration, the information retrieval system can be filled with documents that contain fragment identification numbers. The query itself can also be constructed from fragment

identification numbers.

5 Case Studies

Since our main motivation is the detection of license violations, we set up a database of open source programs. Various programs are matched against the database and the corresponding results are analyzed. We executed our experiments on Java byte-code.

In section 5.1 we explain the performance results obtained. In section 5.2, the quality of the matching procedure is described.

5.1 Performance

In this subsection, we compare the running time of different configurations of a spatial index ($PRTREE$) and the sequential method (SEQ). We execute each configuration with the same query (one program) and the same database. Execution times and the number of distance computations d performed are measured. Also, results for different k and r parameters are shown.

All the tests were done on a database of 340,000 different fragments. The fragment sizes range between 1 and 500 nodes. The query has 1641 fragments and the fragment sizes range between 15 and 30 nodes. The on-disk database size is 30MB. The query size is 100kb. An Ubuntu Linux 6.06 64 bit machine was used to run the experiments. The hardware configuration included two Xeon 64 bit 3.20 GHz processors and 2GB of RAM. The prototype implementation was written in C++.

5.1.1 Optimization by Tree Size

We explored the impact of the naive tree filtering strategy described in section 4.2. Experimentally, we found that for our distance function, tuples of 30 pivots are optimal. We reduced the pivot tuple to 29 elements and added a tree size dimension. Figure 3 shows the effects of using 30 pivots ($PRTREE$, SEQ) or 29 pivots with size filtering ($PRTREE_n$, SEQ_n). For $PRTREE_n$, the performance is not affected. For SEQ_n, the performance is improved substantially as irrelevant trees are discarded. From this point we use the 29 pivot spatial index with one dimension for size.

5.1.2 Using Cache

We added tree caching (section 4.2) to study its effects in performance. Figure 4 shows the effect of this optimization. For $r = 3$, $PRTREE_{ncache}$ outperforms all the other methods. The execution time is 390 seconds. Compare it with SEQ in figure 3 (10 hours).

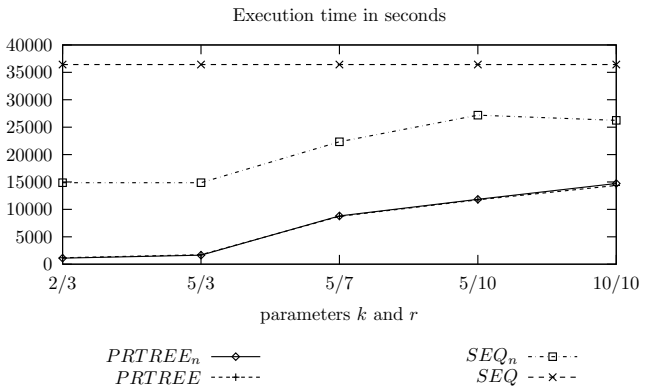

Figure 3. Pruning by tree size

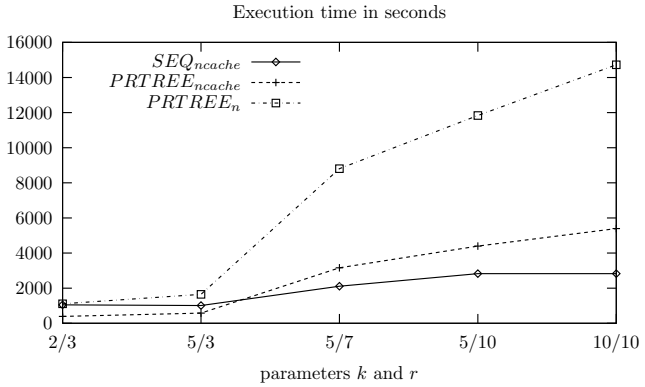

Figure 4. Using Cache

For $r \geq 7$ the sequential method SEQ_{ncache} outperforms the spatial index. Since the range is relatively big, the R-tree is reading randomly almost all the disk pages. The fact that SEQ_{ncache} is reading the fragments sequentially improves performance.

5.1.3 Distance Computation

The number of distance computations performed is shown in figure 5. Because SMAP is calculating a lower bound of d, many pairs of fragments can be discarded. Distance computations are reduced by 95%. The average execution time for SMAP's L_∞ was 0.0004 milliseconds. For d it was 0.005 milliseconds.

5.1.4 Results Summary

For small k and r, a spatial index configured with SMAP and tree caching can improve the performance a hundred times.

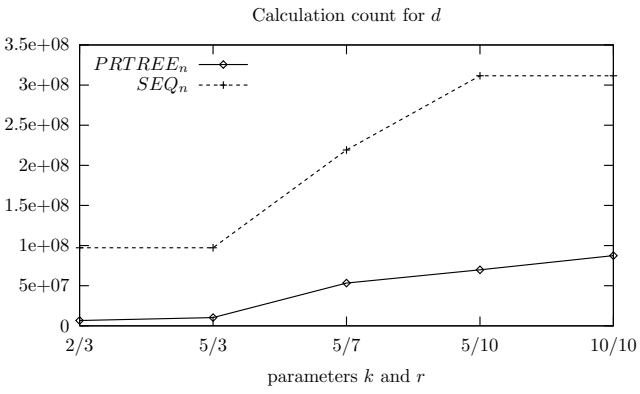

Figure 5. Distance computations

5.2 Program Matching

In this section, we compare the precision of NR and IR ranking techniques. Our main goal is to verify if our techniques can uniquely identify programs in a relatively large database of programs.

A database of Java programs was created by manually downloading binary packages from the Internet and extracting their class files. Also we downloaded Java programs from JPackage[5]. A total of 1878 programs were downloaded. The binary to SSA generation procedure failed for some cases and only 1673 programs were processed successfully. All these programs were fragmented and a database of 1673 applications was created. The fragment sizes range from 1 to 1000. Many types of programs were indexed. From a P2P program called "limewire" to a text editor called "jedit" or an application server called "jboss". The bag of fragments extracted contains 23 million fragments. The set holds 468,505 fragments. Applications with as little as 2 fragments and as many as 255,998 fragments were indexed.

The SSA graphs are extracted with the soot optimization framework [44]. Our prototype is written in Java and is called "Furia".

Obfuscators usually require to have access to program dependencies (libraries). Therefore only JPackage applications were used for the experiments with obfuscated programs. Even though program dependency meta-data is provided by JPackage, not all the programs were obfuscated successfully due to different errors.

Three different sets of queries (pirate programs) were created. All the sets are created from applications that are also in the database. Set A consists of applications with unmodified binaries (just as they were inserted in the database). Set B consists of applications obfuscated with Zelix Klass Master (ZKM)[6] version 4.5. Method, class and field names are shortened. Unused classes, methods or fields are removed. This obfuscator encrypts all the strings embedded in the byte-code and the decryption code has its control flow obfuscated in each method. Control flow obfuscation was set to the maximum level.

Set	Transformation	# of Programs
A	default	1293
B	Zelix	290
C	Sandmark	281

Table 1. Query Sets

Set C was obfuscated with Sandmark 3.4 [2]. We applied 37 of the 39 obfuscation phases available. The phases not applied were "Irreducibility" and "Class Encrypter". The first phase was disabled because soot can have issues with this transformation. The second one was disabled because class encryption attacks are outside the scope of this research. We gave Sandmark an average of 22 seconds per phase to complete and a total of 18 minutes per application. This timeout is set because occasionally, this obfuscator runs for many hours without completing a phase. Obfuscation phase examples are "Insert Opaque Predicates", "Buggy Code", "Inliner" and "BLOAT".

Also we have found that queries whose set of fragments is lower than 100 will not be retrieved successfully. All queries have 100 or more different fragments.

Table 1 summarizes each of the query sets. The number of queries for Zelix and Sandmark are different because Sandmark failed to obfuscate some programs. When we were trying to run a sample of 10 programs obfuscated with Sandmark, only 3 were able to execute.

We consider a *false negative* when no result is returned and we know the query is in the database. A *false positive* occurs when the query that is known to be in the database is not returned within the top n results. A match is considered successful if the program we are looking for appears within the top n results. The value of n is 10 for all the experiments. This value was determined by observing the results of experiment A for IR. The same procedure can be applied when creating new databases. For the following experiments, $r = 3$ and $k = 3$.

5.2.1 Overall Precision

Table 2 shows the result for IR. $\%X$ denotes the accumulated percentage of identifications for the query set X. The value $m(X)$ is the number of matches found for set X. The column n indicates in what position of the candidate list the programs were found. For query sets A and B, identification reaches 100%. In the case of Sandmark, 96% of the applications were identified. The remaining 4% are false positives.

Table 4 displays the results when using the NR scheme. Not even in experiment A this ranking is capable of achieving reliable results. Tables 3 and 5 display a summary of the experiments for IR and NR. No false negatives were found for both ranking techniques. The number of unidentified applications starts in 109 for NR and is only 11 for IR.

5.2.2 Range Effects

For NR, increasing r slightly improves the results in some cases. Table 6 summarizes how different ranges affect the results for NR. Greater ranges improve experiment B at the expense of A. When the range increases, syntactically similar but semantically different fragments can be matched as described in section 3. IR was not improved when changing ranges. By using $r = 0$ none of the three sets of experiments change. The ranking technique IR is robust enough to return correct results even in this case.

5.2.3 Violation Detection

In this section we show how our program can be used in real-life violation detections. The ranking used was IR. A violation is detected when components of the pirate program are inside the database. All the candidates must be included in the top n returned results. In the query set B when matching the application "ccmtools", the following is returned by the system:

```
antlr-2.7.6-1jpp.noarch
antlr-2.7.6-1jpp.noarch.rpm.jpackage
antlr
ccmtools
```

The first two lines are a duplicated entry in the database. The third is another version of "antlr". When exploring the class files of "ccmtools" we found that the program "antlr" is embedded in its class file directory. The first three candidates are at the top of the list because their fragments are less common than the fragments "ccmtools" contains. Lucene gives more weight to matches containing few uncommon fragments than matches of many common fragments. As a side note, the pirate program case study presented in [30] was also successful.

5.2.4 Results Summary

Our experiments show that 96% of the time IR will return correct results for Sandmark (C). For ZKM (B), 100% of the queries were answered correctly. NR is not reliable as only 22% of the queries were correctly identified. The NR scheme fails to achieve acceptable results because all the fragments have the same weight. Also, the distribution of the fragments in the query and in the index are not taken into account.

n	$\%A$	$m(A)$	$\%B$	$m(B)$	$\%C$	$m(C)$
1	97.3	1259	96.8	281	87.5	246
2	98.8	19	99.6	8	90.7	9
3	99.3	7	100	1	92.1	4
4	99.8	6	–	–	93.5	4
5	99.9	1	–	–	94.6	3
6	99.9	0	–	–	94.6	0
7	99.9	0	–	–	95.0	1
8	99.9	0	–	–	95.3	1
9	100	1	–	–	95.7	1
10	–	–	–	–	96.0	1

Table 2. IR ranking

Set	Total	Identified	Not Ident.	False Pos.%
A	1293	1293(100%)	0(0%)	0(0%)
B	290	290(100%)	0(0%)	0(0%)
C	281	270(96%)	11(3.9%)	11(3.9%)

Table 3. IR summary

n	$\%A$	$m(A)$	$\%B$	$m(B)$	$\%C$	$m(C)$
1	18.2	236	4.4	13	9.6	27
2	33.2	194	15.8	33	12.8	9
3	49.1	206	25.5	28	14.2	4
4	59.0	127	33.4	23	16.0	5
5	65.1	80	40.0	19	17.7	5
6	69.9	61	45.5	16	18.1	1
7	73.7	50	51.0	16	19.5	4
8	77.4	48	54.1	9	20.6	3
9	80.6	41	58.2	12	21.3	2
10	83.6	38	62.4	12	22.7	4

Table 4. NR ranking

Set	Total	Identified	Not Ident.	False Pos.%
A	1293	1081(83.6%)	212(16.3%)	212(16.3%)
B	290	181(62.4%)	109(37.5%)	109(37.5%)
C	281	64(22.7%)	217(77.2%)	217(77.2%)

Table 5. NR summary

Set	$r = 3$	$r = 7$	$r = 10$	$r = 15$
A	1081	1076	1075	1074
B	181	186	188	191
C	64	64	65	64

Table 6. Identified applications when changing r (NR)

6 Related Research

The main obstacle present in the field of program matching is the fact that checking the semantic equivalence between two programs is undecidable. Pattern matching approaches cannot be used because different compilers produce different binary programs. Even the same compiler can generate different outputs depending on the optimization or code generation flags used. Furthermore, the fact that obfuscator programs can mangle the binary enough to prevent reverse engineering[17], imposes a challenge. Also, several important static analysis problems are undecidable or computationally hard[27, 31].

Nevertheless, a result by Barak *et al.* [10] proves that in general program obfuscation is impossible. This leads us to believe that a computationally bounded obfuscator will not be able to obfuscate a program completely. Approximation attempts have been successful in restricted domains[45, 23].

In section 6.1 we introduce different tree matching techniques. In section 6.2, several approaches that can be used for program matching are described.

6.1 Tree Matching

Several approaches exist to match trees. Some techniques focus on improving the distance computation of a pair of trees. Weimann *et al.*[19] implemented a tree-edit distance algorithm with complexity $O(n^3)$. Touzet[42] proposed an $O(nk^3)$ algorithm for trees with k or less errors. Chawathe[13] proposed a distance function that is optimized for external storage. Augsten *et al.*[7] introduced an $O(n)$ distance that is more sensitive to structure changes.

Other techniques try to minimize the amount of distance computations[48, 21, 32]. The approach followed by Kailing[24] is of particular interest. This technique generates vectors of tree features. Those vectors can be indexed in a spatial index. In their paper, they used the X-tree[12].

A more recent work by Yang *et al.* has outperformed Kailing's work by combining n-grams and an optimal filter and refine technique proposed by Seidl and Kriegel [34]. The techniques employed by Kailing and Yang, are based on the standard definition of tree-edit distance and they might not work well if the distance function is changed. To achieve our main objective, we might need to use different tree-edit distance definitions. Therefore it was necessary to find a more general approach. The M-tree[15] is general enough to satisfy our needs, however a combination of an R-tree and a dimension reduction technique called SMAP[35] performs much better in practice. We experimentally confirmed this for trees.

6.2 Program Matching

A first approach is to match programs by the strings they have in common. The "strings" technique for detecting violations is not robust enough, as obfuscators can encrypt the strings embedded in a program. Baker[9] introduced some techniques based on adapting existing source code similarity analysis tools. A prototype was implemented and validated for Java byte-code programs. The author recognizes that those techniques cannot work with obfuscated programs as they are sensitive to the order in which instructions occur.

Watermarking is a technique that embeds stealthy information that identifies the program author (either in a static [29] or dynamic [16] manner). Since the source code of the applications we want to protect is open and available to anyone, this technique cannot be employed.

A technique called *clone detection* [11, 36] is used to detect duplicate fragments of code in source code to reduce maintainability problems in software projects. Automatic tools for measuring *software similarity* [33, 28] have been also created to perform *plagiarism detection*. Nevertheless, the source code for the application that illegally contains libre software is not available in our problem setting.

Birth-marking [38, 39, 40] is a technique that extracts unique and "native" characteristics of every class file. The approach relies heavily on class information that is only found in higher level binary representations such as the Java Byte-code. The technique has been tested in very small databases only. Our approach is more general because it only requires that the binary program can be abstracted into a series of assignment statements.

A dynamic approach that collects birth-marks on system API access patterns was proposed [41]. The technique works by analyzing the sequence and the frequency of function calls of a program. This approach has been validated only with small datasets and it is not clear how well it scales.

Malicious code detection [14] is a technique used to find obfuscated viruses in programs. It works on annotated Control Flow graphs using a Malicious Code Automaton (MCA) that can be seen as an extended regular expression. In this work the matching is done by finding if the malicious pattern described by the MCA is in the annotated CFG. The approach is powerful but an automatic MCA generator algorithm is not available. As viruses are relatively small, a manual construction approach is feasible. Building an automatic MCA generator that takes into account all the possible transformations made by all the compilers and obfuscators seems much more complicated than the proposal we are presenting here.

Dullien and Rolles[20] introduced a method that creates optimal isomorphisms between sets of instructions. The method recursively finds similar pairs of graphs at the pro-

gram, control flow graph, and basic block level. The authors point out that in-lining of complex functions can affect the matching performance. As explained in section 4.3, our technique can match fragments independently of where they are located and therefore in-lining attacks are less likely to affect the results.

Eric Raymond's program "comparator"[3] works by partitioning a program into small blocks that are matched afterwards. An important feature is that the position of the blocks is irrelevant in the matching process. This is similar to our technique, however Raymond's approach is vulnerable to trivial renaming of variables and obfuscation. Our approach solves this by removing variable names altogether.

7 Conclusions and Future Work

We have experimentally demonstrated how spatial indexes can substantially improve the performance of an approximate program matcher. We also have improved identification results by using information retrieval techniques. With the techniques presented in this paper, it is possible to identify programs even if they are heavily obfuscated.

To take advantage of the locality characteristics that the sequential method enjoys, it is necessary to sort the fragments of the query in a way that optimizes disk caching. Tree caching has an important effect in performance. A cache manager that disposes less frequently accessed fragments to conserve memory has to be implemented for the R-tree. All these improvements shall be explored in the future.

Further improvements are required for eliminating false positives. The IR ranking method has information that can be used for this purpose. We are currently working on this. A possible way of achieving this would be to adapt more intensive techniques[26, 43, 20] to binary programs. This is feasible because with the results presented in this paper, only n programs have to be processed. As long as the fragmentation technique employed generates unique fingerprints, n should be small. In our case studies, $n = 10$. A method for rejecting pairs of fragments that are syntactically close but semantically different is required.

The matching technique we have presented here is simplified in the sense that it does not use unexpanded fragment references to make sure that the relationships among fragments are preserved. Using this information may introduce false negatives. Nevertheless, the ability to exhaustively confirm a candidate is desirable in our application domain. An important feature we have not implemented yet is fragment normalization. We believe that it may be possible to automatically or semi-automatically learn fragment normalization rules. Future research shall focus on all these aspects.

Other possible applications of this research include approximate malware detection, and low-level functionality duplication detection to reduce maintainability problems.

8 Acknowledgements

This work was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology (grant # 040506). We would like to thank our reviewers for all their extremely valuable input.

References

[1] http://lucene.apache.org/.
[2] http://sandmark.cs.arizona.edu/index.html.
[3] http://www.catb.org/~esr/comparator/.
[4] http://www.fsf.org/licensing/licenses/ and http://gpl-violations.org/.
[5] http://www.jpackage.org/.
[6] http://www.zelix.com.
[7] N. Augsten, M. Bhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Databases*, pages 301–312. VLDB Endowment, 2005.
[8] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[9] B. S. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *Proc. of Usenix Annual Technical Conf.*, pages 179–190, 1998.
[10] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO*, pages 1–18, 2001.
[11] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
[12] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.
[13] S. S. Chawathe. Comparing hierarchical data in external memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
[14] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. *12th USENIX Security Symposium*, pages 169–186, 2003.
[15] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.
[16] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, pages 311–324, 1999.

[17] C. S. Collberg, C. D. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[19] E. Demaine, S. Mosez, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *www.mit.edu/ oweimann/*, 2005.

[20] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *SSTIC*, 2005.

[21] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate xml joins. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 287–298, New York, NY, USA, 2002. ACM Press.

[22] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 1990. ACM Press.

[24] K. Kailing, H.-P. Kriegel, S. Schoenauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *EDBT 2004*, pages 676–693, 2004.

[25] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. Interprocedural static slicing of binary executables. *SCAM*, 00:118, 2003.

[26] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Proceedings of the 8th International Symposium on Static Analysis*, 2126:40–60, 2001.

[27] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[28] W. M. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin*, 28:130–134, 1996.

[29] A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii. A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*, pages 191–197, 2000.

[30] A. Müller and T. Shinohara. On approximate matching of programs for protecting libre software. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative Research*, pages 275–289, New York, NY, USA, 2006. ACM Press.

[31] E. M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–230, New York, NY, USA, 1981. ACM Press.

[32] N. Ohkura, K. Hirata, T. Kuboyama, and M. Harao. The q-gram distance for ordered unlabeled trees. In *Discovery Science (8th international conference)*, pages 189–202, 2004.

[33] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarism among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[34] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 154–165, New York, NY, USA, 1998. ACM Press.

[35] T. Shinohara and H. Ishizaka. On dimension reduction mappings for approximate retrieval of multi-dimensional data. In *Progress in Discovery Science, Final Report of the Japanese Discovery Science Project*, pages 224–231, London, UK, 2002. Springer-Verlag.

[36] K. T., K. S., and I. K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering 28(7)*, pages 654–670, 2002.

[37] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[38] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. Detecting the theft of programs using birthmarks. *Information Science Technical Report*, 2003.

[39] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. A method for extracting program fingerprints from java class files. *The Institute of Electronics, Information and Communication Engineers Technical Report*, ISEC2003-29:127–133, 2003.

[40] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto. Java birthmarks: Detecting the software theft. *IEICE Trans Inf Syst*, E88-D(9):2148–2158, 2005.

[41] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. *International Symposium on Future Software Technology 2004 (ISFST 2004)*, 2004.

[42] H. Touzet. A linear tree edit distance algorithm for similar ordered trees. In *Lecture Notes in Computer Science*, volume 3537, pages 334–345, 2005.

[43] F. Umemori, K. Konda, R. Yokomori, and K. Inoue. Design and implementation of bytecode-based java slicing system. *SCAM*, 00:108–117, 2003.

[44] R. Vall-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. *CASCON99*, pages 13–24, 1999.

[45] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations on low-level program representations. *Sci. Comput. Program.*, 52(1-3):257–280, 2004.

[46] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society.

[47] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, pages 352–357, 1984.

[48] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 754–765, 2005.